

The Java API Math, Text and Time Sources

Marco Mendieta Parihuancollo

Versión 036MPACAGAh1 (2025-10-15)

Secuencia 1: Source code

root: C:\Program Files\Java\jdk1.8.0_131\src.zip

Section	Package (8)	Files (135)	Lines (91807)
Math	java.math	8	13175
Text	java.text	34	22113
Text Spi	java.text.spi	6	529
Time	java.time	21	22456
Time Chrono	java.time.chrono	27	12950
Time Format	java.time.format	14	10601
Time Temporal	java.time.temporal	17	6763
Time Zone	java.time.zone	8	3220

Líneas de código: 91807 (1765 pages)

Contents

1	Math	1
1.1	BigDecimal.java	1
1.2	BigInteger.java	102
1.3	BitSieve.java	189
1.4	MathContext.java	193
1.5	MutableBigInteger.java	199
1.6	RoundingMode.java	242
1.7	SignedMutableBigInteger.java	249
1.8	package-info.java	252
2	Text	253
2.1	Annotation.java	253
2.2	AttributedCharacterIterator.java	255
2.3	AttributedString.java	260
2.4	Bidi.java	282
2.5	BreakIterator.java	288
2.6	CalendarBuilder.java	300
2.7	CharacterIterator.java	304
2.8	CharacterIteratorFieldDelegate.java	307
2.9	ChoiceFormat.java	310
2.10	CollationElementIterator.java	322
2.11	CollationKey.java	337

2.12	Collator.java	340
2.13	DateFormat.java	350
2.14	DateFormatSymbols.java	369
2.15	DecimalFormat.java	386
2.16	DecimalFormatSymbols.java	465
2.17	DigitList.java	482
2.18	DontCareFieldPosition.java	498
2.19	EntryPair.java	499
2.20	FieldPosition.java	500
2.21	Format.java	506
2.22	MergeCollation.java	514
2.23	MessageFormat.java	520
2.24	Normalizer.java	551
2.25	NumberFormat.java	554
2.26	ParseException.java	578
2.27	ParsePosition.java	579
2.28	PatternEntry.java	582
2.29	RBCollationTables.java	588
2.30	RBTableBuilder.java	594
2.31	RuleBasedCollationKey.java	606
2.32	RuleBasedCollator.java	608
2.33	SimpleDateFormat.java	623
2.34	StringCharacterIterator.java	669
3	Text Spi	675
3.1	BreakIteratorProvider.java	675
3.2	CollatorProvider.java	677
3.3	DateFormatProvider.java	678
3.4	DateFormatSymbolsProvider.java	680
3.5	DecimalFormatSymbolsProvider.java	681
3.6	NumberFormatProvider.java	683
4	Time	685
4.1	Clock.java	685
4.2	DateTimeException.java	697
4.3	DayOfWeek.java	699
4.4	Duration.java	708
4.5	Instant.java	734
4.6	LocalDate.java	761
4.7	LocalDateTime.java	801
4.8	LocalTime.java	839
4.9	Month.java	872
4.10	MonthDay.java	884
4.11	OffsetDateTime.java	899
4.12	OffsetTime.java	937
4.13	Period.java	964
4.14	Ser.java	985
4.15	Year.java	991
4.16	YearMonth.java	1012
4.17	ZoneId.java	1036
4.18	ZoneOffset.java	1049
4.19	ZoneRegion.java	1064
4.20	ZonedDateTime.java	1068

4.21	package-info.java	1112
5	Time Chrono	1118
5.1	AbstractChronology.java	1118
5.2	ChronoLocalDate.java	1133
5.3	ChronoLocalDateImpl.java	1148
5.4	ChronoLocalDateTime.java	1157
5.5	ChronoLocalDateTimeImpl.java	1169
5.6	ChronoPeriod.java	1178
5.7	ChronoPeriodImpl.java	1185
5.8	ChronoZonedDateTime.java	1192
5.9	ChronoZonedDateTimeImpl.java	1205
5.10	Chronology.java	1213
5.11	Era.java	1228
5.12	HijrahChronology.java	1234
5.13	HijrahDate.java	1255
5.14	HijrahEra.java	1268
5.15	IsoChronology.java	1271
5.16	IsoEra.java	1283
5.17	JapaneseChronology.java	1286
5.18	JapaneseDate.java	1297
5.19	JapaneseEra.java	1311
5.20	MinguoChronology.java	1318
5.21	MinguoDate.java	1325
5.22	MinguoEra.java	1335
5.23	Ser.java	1338
5.24	ThaiBuddhistChronology.java	1344
5.25	ThaiBuddhistDate.java	1352
5.26	ThaiBuddhistEra.java	1362
5.27	package-info.java	1365
6	Time Format	1368
6.1	DateTimeFormatter.java	1368
6.2	DateTimeFormatterBuilder.java	1410
6.3	DateTimeParseException.java	1497
6.4	DateTimeParseException.java	1505
6.5	DateTimePrintContext.java	1508
6.6	DateTimeTextProvider.java	1514
6.7	DecimalStyle.java	1525
6.8	FormatStyle.java	1532
6.9	Parsed.java	1534
6.10	ResolverStyle.java	1548
6.11	SignStyle.java	1550
6.12	TextStyle.java	1553
6.13	ZoneName.java	1556
6.14	package-info.java	1571
7	Time Temporal	1573
7.1	ChronoField.java	1573
7.2	ChronoUnit.java	1588
7.3	IsoFields.java	1593
7.4	JulianFields.java	1607
7.5	Temporal.java	1613

7.6	TemporalAccessor.java	1622
7.7	TemporalAdjuster.java	1628
7.8	TemporalAdjusters.java	1631
7.9	TemporalAmount.java	1640
7.10	TemporalField.java	1645
7.11	TemporalQueries.java	1652
7.12	TemporalQuery.java	1660
7.13	TemporalUnit.java	1663
7.14	UnsupportedTemporalTypeException.java	1668
7.15	ValueRange.java	1670
7.16	WeekFields.java	1679
7.17	package-info.java	1701
8	Time Zone	1705
8.1	Ser.java	1705
8.2	TzdbZoneRulesProvider.java	1710
8.3	ZoneOffsetTransition.java	1714
8.4	ZoneOffsetTransitionRule.java	1723
8.5	ZoneRules.java	1735
8.6	ZoneRulesException.java	1755
8.7	ZoneRulesProvider.java	1757
8.8	package-info.java	1766

1 Math (java.math package)

1.1 BigDecimal.java

Secuencia 2: java/math/BigDecimal.java

```

1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */

```



```
25
26 /*
27  * Portions Copyright IBM Corporation, 2001. All Rights Reserved.
28  */
29
30 package java.math;
31
32 import java.util.Arrays;
33 import static java.math.BigInteger.LONG_MASK;
34
35 /**
36  * Immutable, arbitrary-precision signed decimal numbers. A
37  * {@code BigDecimal} consists of an arbitrary precision integer
38  * unscaled value and a 32-bit integer scale. If zero
39  * or positive, the scale is the number of digits to the right of the
40  * decimal point. If negative, the unscaled value of the number is
41  * multiplied by ten to the power of the negation of the scale. The
42  * value of the number represented by the {@code BigDecimal} is
43  * therefore  $\text{unscaledValue} \times 10^{-\text{scale}}$ .
44  *
45  * 

The {@code BigDecimal} class provides operations for
46  * arithmetic, scale manipulation, rounding, comparison, hashing, and
47  * format conversion. The {@link #toString} method provides a
48  * canonical representation of a {@code BigDecimal}.


49  *
50  * 

The {@code BigDecimal} class gives its user complete control
51  * over rounding behavior. If no rounding mode is specified and the
52  * exact result cannot be represented, an exception is thrown;
53  * otherwise, calculations can be carried out to a chosen precision
54  * and rounding mode by supplying an appropriate {@link MathContext}
55  * object to the operation. In either case, eight rounding
56  * modes are provided for the control of rounding. Using the
57  * integer fields in this class (such as {@link #ROUND_HALF_UP}) to
58  * represent rounding mode is largely obsolete; the enumeration values
59  * of the {@code RoundingMode} enum, (such as {@link
60  * RoundingMode#HALF_UP}) should be used instead.


61  *
62  * 

When a {@code MathContext} object is supplied with a precision
63  * setting of 0 (for example, {@link MathContext#UNLIMITED}),
64  * arithmetic operations are exact, as are the arithmetic methods
65  * which take no {@code MathContext} object. (This is the only
66  * behavior that was supported in releases prior to 5.) As a
67  * corollary of computing the exact result, the rounding mode setting
68  * of a {@code MathContext} object with a precision setting of 0 is
69  * not used and thus irrelevant. In the case of divide, the exact
70  * quotient could have an infinitely long decimal expansion; for
71  * example, 1 divided by 3. If the quotient has a nonterminating
72  * decimal expansion and the operation is specified to return an exact
73  * result, an ArithmeticException is thrown. Otherwise, the
74  * exact result of the division is returned, as done for other
75  * operations.


76  *
77  * 

When the precision setting is not 0, the rules of


```

```

78 * {@code BigDecimal} arithmetic are broadly compatible with selected
79 * modes of operation of the arithmetic defined in ANSI X3.274-1996
80 * and ANSI X3.274-1996/AM 1-2000 (section 7.4). Unlike those
81 * standards, {@code BigDecimal} includes many rounding modes, which
82 * were mandatory for division in {@code BigDecimal} releases prior
83 * to 5. Any conflicts between these ANSI standards and the
84 * {@code BigDecimal} specification are resolved in favor of
85 * {@code BigDecimal}.
86 *
87 * <p>Since the same numerical value can have different
88 * representations (with different scales), the rules of arithmetic
89 * and rounding must specify both the numerical result and the scale
90 * used in the result's representation.
91 *
92 *
93 * <p>In general the rounding modes and precision setting determine
94 * how operations return results with a limited number of digits when
95 * the exact result has more digits (perhaps infinitely many in the
96 * case of division) than the number of digits returned.
97 *
98 * First, the
99 * total number of digits to return is specified by the
100 * {@code MathContext}'s {@code precision} setting; this determines
101 * the result's <i>precision</i>. The digit count starts from the
102 * leftmost nonzero digit of the exact result. The rounding mode
103 * determines how any discarded trailing digits affect the returned
104 * result.
105 *
106 * <p>For all arithmetic operators , the operation is carried out as
107 * though an exact intermediate result were first calculated and then
108 * rounded to the number of digits specified by the precision setting
109 * (if necessary), using the selected rounding mode. If the exact
110 * result is not returned, some digit positions of the exact result
111 * are discarded. When rounding increases the magnitude of the
112 * returned result, it is possible for a new digit position to be
113 * created by a carry propagating to a leading {@literal "9"} digit.
114 * For example, rounding the value 999.9 to three digits rounding up
115 * would be numerically equal to one thousand, represented as
116 * 100&times;10<sup>1</sup>. In such cases, the new {@literal "1"} is
117 * the leading digit position of the returned result.
118 *
119 * <p>Besides a logical exact result, each arithmetic operation has a
120 * preferred scale for representing a result. The preferred
121 * scale for each operation is listed in the table below.
122 *
123 * <table border>
124 * <caption><b>Preferred Scales for Results of Arithmetic Operations
125 * </b></caption>
126 * <tr><th>Operation</th><th>Preferred Scale of Result</th></tr>
127 * <tr><td>Add</td><td>max(addend.scale(), augend.scale())</td>
128 * <tr><td>Subtract</td><td>max(minuend.scale(), subtrahend.scale())</td>
129 * <tr><td>Multiply</td><td>multiplier.scale() + multiplicand.scale()</td>
130 * <tr><td>Divide</td><td>dividend.scale() - divisor.scale()</td>

```

```

131 * </table>
132 *
133 * These scales are the ones used by the methods which return exact
134 * arithmetic results; except that an exact divide may have to use a
135 * larger scale since the exact result may have more digits. For
136 * example, {@code 1/32} is {@code 0.03125}.
137 *
138 * <p>Before rounding, the scale of the logical exact intermediate
139 * result is the preferred scale for that operation. If the exact
140 * numerical result cannot be represented in {@code precision}
141 * digits, rounding selects the set of digits to return and the scale
142 * of the result is reduced from the scale of the intermediate result
143 * to the least scale which can represent the {@code precision}
144 * digits actually returned. If the exact result can be represented
145 * with at most {@code precision} digits, the representation
146 * of the result with the scale closest to the preferred scale is
147 * returned. In particular, an exactly representable quotient may be
148 * represented in fewer than {@code precision} digits by removing
149 * trailing zeros and decreasing the scale. For example, rounding to
150 * three digits using the {@link RoundingMode#FLOOR floor}
151 * rounding mode, <br>
152 *
153 * {@code 19/100 = 0.19 // integer=19, scale=2} <br>
154 *
155 * but<br>
156 *
157 * {@code 21/110 = 0.190 // integer=190, scale=3} <br>
158 *
159 * <p>Note that for add, subtract, and multiply, the reduction in
160 * scale will equal the number of digit positions of the exact result
161 * which are discarded. If the rounding causes a carry propagation to
162 * create a new high-order digit position, an additional digit of the
163 * result is discarded than when no new digit position is created.
164 *
165 * <p>Other methods may have slightly different rounding semantics.
166 * For example, the result of the {@code pow} method using the
167 * {@link #pow(int, MathContext) specified algorithm} can
168 * occasionally differ from the rounded mathematical result by more
169 * than one unit in the last place, one <i>{@link #ulp() ulp}</i>.
170 *
171 * <p>Two types of operations are provided for manipulating the scale
172 * of a {@code BigDecimal}: scaling/rounding operations and decimal
173 * point motion operations. Scaling/rounding operations ({@link
174 * #setScale setScale} and {@link #round round}) return a
175 * {@code BigDecimal} whose value is approximately (or exactly) equal
176 * to that of the operand, but whose scale or precision is the
177 * specified value; that is, they increase or decrease the precision
178 * of the stored number with minimal effect on its value. Decimal
179 * point motion operations ({@link #movePointLeft movePointLeft} and
180 * {@link #movePointRight movePointRight}) return a
181 * {@code BigDecimal} created from the operand by moving the decimal
182 * point a specified distance in the specified direction.
183 *

```

```

184 * <p>For the sake of brevity and clarity, pseudo-code is used
185 * throughout the descriptions of {@code BigDecimal} methods. The
186 * pseudo-code expression {@code (i + j)} is shorthand for "a
187 * {@code BigDecimal} whose value is that of the {@code BigDecimal}
188 * {@code i} added to that of the {@code BigDecimal}
189 * {@code j}." The pseudo-code expression {@code (i == j)} is
190 * shorthand for "{@code true} if and only if the
191 * {@code BigDecimal} {@code i} represents the same value as the
192 * {@code BigDecimal} {@code j}." Other pseudo-code expressions
193 * are interpreted similarly. Square brackets are used to represent
194 * the particular {@code BigInteger} and scale pair defining a
195 * {@code BigDecimal} value; for example [19, 2] is the
196 * {@code BigDecimal} numerically equal to 0.19 having a scale of 2.
197 *
198 * <p>Note: care should be exercised if {@code BigDecimal} objects
199 * are used as keys in a {@link java.util.SortedMap SortedMap} or
200 * elements in a {@link java.util.SortedSet SortedSet} since
201 * {@code BigDecimal}'s <i>natural ordering</i> is <i>inconsistent
202 * with equals</i>. See {@link Comparable}, {@link
203 * java.util.SortedMap} or {@link java.util.SortedSet} for more
204 * information.
205 *
206 * <p>All methods and constructors for this class throw
207 * {@code NullPointerException} when passed a {@code null} object
208 * reference for any input parameter.
209 *
210 * @see      BigInteger
211 * @see      MathContext
212 * @see      RoundingMode
213 * @see      java.util.SortedMap
214 * @see      java.util.SortedSet
215 * @author    Josh Bloch
216 * @author    Mike Cowlishaw
217 * @author    Joseph D. Darcy
218 * @author    Sergey V. Kuksenkov
219 */
220 public class BigDecimal extends Number implements Comparable<BigDecimal> {
221     /**
222      * The unscaled value of this BigDecimal, as returned by {@link
223      * #unscaledValue}.
224      *
225      * @serial
226      * @see #unscaledValue
227      */
228     private final BigInteger intVal;
229
230     /**
231      * The scale of this BigDecimal, as returned by {@link #scale}.
232      *
233      * @serial
234      * @see #scale
235      */
236     private final int scale; // Note: this may have any value, so

```

```

237         // calculations must be done in longs
238
239     /**
240     * The number of decimal digits in this BigDecimal, or 0 if the
241     * number of digits are not known (lookaside information). If
242     * nonzero, the value is guaranteed correct. Use the precision()
243     * method to obtain and set the value if it might be 0. This
244     * field is mutable until set nonzero.
245     *
246     * @since 1.5
247     */
248     private transient int precision;
249
250     /**
251     * Used to store the canonical string representation, if computed.
252     */
253     private transient String stringCache;
254
255     /**
256     * Sentinel value for {@link #intCompact} indicating the
257     * significand information is only available from {@code intVal}.
258     */
259     static final long INFLATED = Long.MIN_VALUE;
260
261     private static final BigInteger INFLATED_BIGINT = BigInteger.valueOf(INFLATED);
262
263     /**
264     * If the absolute value of the significand of this BigDecimal is
265     * less than or equal to {@code Long.MAX_VALUE}, the value can be
266     * compactly stored in this field and used in computations.
267     */
268     private final transient long intCompact;
269
270     // All 18-digit base ten strings fit into a long; not all 19-digit
271     // strings will
272     private static final int MAX_COMPACT_DIGITS = 18;
273
274     /* Appease the serialization gods */
275     private static final long serialVersionUID = 6108874887143696463L;
276
277     private static final ThreadLocal<StringBuilderHelper>
278         threadLocalStringBuilderHelper = new ThreadLocal<StringBuilderHelper>() {
279         @Override
280         protected StringBuilderHelper initialValue() {
281             return new StringBuilderHelper();
282         }
283     };
284
285     // Cache of common small BigDecimal values.
286     private static final BigDecimal zeroThroughTen[] = {
287         new BigDecimal(BigInteger.ZERO, 0, 0, 1),
288         new BigDecimal(BigInteger.ONE, 1, 0, 1),
289         new BigDecimal(BigInteger.valueOf(2), 2, 0, 1),

```

```

290     new BigDecimal(BigInteger.valueOf(3), 3, 0, 1),
291     new BigDecimal(BigInteger.valueOf(4), 4, 0, 1),
292     new BigDecimal(BigInteger.valueOf(5), 5, 0, 1),
293     new BigDecimal(BigInteger.valueOf(6), 6, 0, 1),
294     new BigDecimal(BigInteger.valueOf(7), 7, 0, 1),
295     new BigDecimal(BigInteger.valueOf(8), 8, 0, 1),
296     new BigDecimal(BigInteger.valueOf(9), 9, 0, 1),
297     new BigDecimal(BigInteger.TEN, 10, 0, 2),
298 };
299
300 // Cache of zero scaled by 0 - 15
301 private static final BigDecimal[] ZERO_SCALED_BY = {
302     zeroThroughTen[0],
303     new BigDecimal(BigInteger.ZERO, 0, 1, 1),
304     new BigDecimal(BigInteger.ZERO, 0, 2, 1),
305     new BigDecimal(BigInteger.ZERO, 0, 3, 1),
306     new BigDecimal(BigInteger.ZERO, 0, 4, 1),
307     new BigDecimal(BigInteger.ZERO, 0, 5, 1),
308     new BigDecimal(BigInteger.ZERO, 0, 6, 1),
309     new BigDecimal(BigInteger.ZERO, 0, 7, 1),
310     new BigDecimal(BigInteger.ZERO, 0, 8, 1),
311     new BigDecimal(BigInteger.ZERO, 0, 9, 1),
312     new BigDecimal(BigInteger.ZERO, 0, 10, 1),
313     new BigDecimal(BigInteger.ZERO, 0, 11, 1),
314     new BigDecimal(BigInteger.ZERO, 0, 12, 1),
315     new BigDecimal(BigInteger.ZERO, 0, 13, 1),
316     new BigDecimal(BigInteger.ZERO, 0, 14, 1),
317     new BigDecimal(BigInteger.ZERO, 0, 15, 1),
318 };
319
320 // Half of Long.MIN_VALUE & Long.MAX_VALUE.
321 private static final long HALF_LONG_MAX_VALUE = Long.MAX_VALUE / 2;
322 private static final long HALF_LONG_MIN_VALUE = Long.MIN_VALUE / 2;
323
324 // Constants
325 /**
326  * The value 0, with a scale of 0.
327  *
328  * @since 1.5
329  */
330 public static final BigDecimal ZERO =
331     zeroThroughTen[0];
332
333 /**
334  * The value 1, with a scale of 0.
335  *
336  * @since 1.5
337  */
338 public static final BigDecimal ONE =
339     zeroThroughTen[1];
340
341 /**
342  * The value 10, with a scale of 0.

```

```

343     *
344     * @since 1.5
345     */
346     public static final BigDecimal TEN =
347         zeroThroughTen[10];
348
349     // Constructors
350
351     /**
352     * Trusted package private constructor.
353     * Trusted simply means if val is INFLATED, intVal could not be null and
354     * if intVal is null, val could not be INFLATED.
355     */
356     BigDecimal(BigInteger intVal, long val, int scale, int prec) {
357         this.scale = scale;
358         this.precision = prec;
359         this.intCompact = val;
360         this.intVal = intVal;
361     }
362
363     /**
364     * Translates a character array representation of a
365     * {@code BigDecimal} into a {@code BigDecimal}, accepting the
366     * same sequence of characters as the {@link #BigDecimal(String)}
367     * constructor, while allowing a sub-array to be specified.
368     *
369     * <p>Note that if the sequence of characters is already available
370     * within a character array, using this constructor is faster than
371     * converting the {@code char} array to string and using the
372     * {@code BigDecimal(String)} constructor .
373     *
374     * @param in {@code char} array that is the source of characters.
375     * @param offset first character in the array to inspect.
376     * @param len number of characters to consider.
377     * @throws NumberFormatException if {@code in} is not a valid
378     *         representation of a {@code BigDecimal} or the defined subarray
379     *         is not wholly within {@code in}.
380     * @since 1.5
381     */
382     public BigDecimal(char[] in, int offset, int len) {
383         this(in, offset, len, MathContext.UNLIMITED);
384     }
385
386     /**
387     * Translates a character array representation of a
388     * {@code BigDecimal} into a {@code BigDecimal}, accepting the
389     * same sequence of characters as the {@link #BigDecimal(String)}
390     * constructor, while allowing a sub-array to be specified and
391     * with rounding according to the context settings.
392     *
393     * <p>Note that if the sequence of characters is already available
394     * within a character array, using this constructor is faster than
395     * converting the {@code char} array to string and using the

```



```

396 * {@code BigDecimal(String)} constructor .
397 *
398 * @param in {@code char} array that is the source of characters.
399 * @param offset first character in the array to inspect.
400 * @param len number of characters to consider..
401 * @param mc the context to use.
402 * @throws ArithmeticException if the result is inexact but the
403 *         rounding mode is {@code UNNECESSARY}.
404 * @throws NumberFormatException if {@code in} is not a valid
405 *         representation of a {@code BigDecimal} or the defined subarray
406 *         is not wholly within {@code in}.
407 * @since 1.5
408 */
409 public BigDecimal(char[] in, int offset, int len, MathContext mc) {
410     // protect against huge length.
411     if (offset + len > in.length || offset < 0)
412         throw new NumberFormatException("Bad offset or len arguments for char[] input.");
413     // This is the primary string to BigDecimal constructor; all
414     // incoming strings end up here; it uses explicit (inline)
415     // parsing for speed and generates at most one intermediate
416     // (temporary) object (a char[] array) for non-compact case.
417
418     // Use locals for all fields values until completion
419     int prec = 0;           // record precision value
420     int scl = 0;           // record scale value
421     long rs = 0;           // the compact value in long
422     BigInteger rb = null;   // the inflated value in BigInteger
423     // use array bounds checking to handle too-long, len == 0,
424     // bad offset, etc.
425     try {
426         // handle the sign
427         boolean isneg = false;           // assume positive
428         if (in[offset] == '-') {
429             isneg = true;                 // leading minus means negative
430             offset++;
431             len--;
432         } else if (in[offset] == '+') { // leading + allowed
433             offset++;
434             len--;
435         }
436
437         // should now be at numeric part of the significand
438         boolean dot = false;             // true when there is a '.'
439         long exp = 0;                     // exponent
440         char c;                           // current character
441         boolean isCompact = (len <= MAX_COMPACT_DIGITS);
442         // integer significand array & idx is the index to it. The array
443         // is ONLY used when we can't use a compact representation.
444         int idx = 0;
445         if (isCompact) {
446             // First compact case, we need not to preserve the character
447             // and we can just compute the value in place.
448             for (; len > 0; offset++, len--) {

```



```

449         c = in[offset];
450         if ((c == '0')) { // have zero
451             if (prec == 0)
452                 prec = 1;
453             else if (rs != 0) {
454                 rs *= 10;
455                 ++prec;
456             } // else digit is a redundant leading zero
457             if (dot)
458                 ++scl;
459         } else if ((c >= '1' && c <= '9')) { // have digit
460             int digit = c - '0';
461             if (prec != 1 || rs != 0)
462                 ++prec; // prec unchanged if preceded by 0s
463             rs = rs * 10 + digit;
464             if (dot)
465                 ++scl;
466         } else if (c == '.') { // have dot
467             // have dot
468             if (dot) // two dots
469                 throw new NumberFormatException();
470             dot = true;
471         } else if (Character.isDigit(c)) { // slow path
472             int digit = Character.digit(c, 10);
473             if (digit == 0) {
474                 if (prec == 0)
475                     prec = 1;
476                 else if (rs != 0) {
477                     rs *= 10;
478                     ++prec;
479                 } // else digit is a redundant leading zero
480             } else {
481                 if (prec != 1 || rs != 0)
482                     ++prec; // prec unchanged if preceded by 0s
483                 rs = rs * 10 + digit;
484             }
485             if (dot)
486                 ++scl;
487         } else if ((c == 'e') || (c == 'E')) {
488             exp = parseExp(in, offset, len);
489             // Next test is required for backwards compatibility
490             if ((int) exp != exp) // overflow
491                 throw new NumberFormatException();
492             break; // [saves a test]
493         } else {
494             throw new NumberFormatException();
495         }
496     }
497     if (prec == 0) // no digits found
498         throw new NumberFormatException();
499     // Adjust scale if exp is not zero.
500     if (exp != 0) { // had significant exponent
501         scl = adjustScale(scl, exp);

```

```

502     }
503     rs = isneg ? -rs : rs;
504     int mcp = mc.precision;
505     int drop = prec - mcp; // prec has range [1, MAX_INT], mcp has range [0, MAX_INT];
506                          // therefore, this subtract cannot overflow
507     if (mcp > 0 && drop > 0) { // do rounding
508         while (drop > 0) {
509             scl = checkScaleNonZero((long) scl - drop);
510             rs = divideAndRound(rs, LONG_TEN_POWERS_TABLE[drop], mc.roundingMode.
                    oldMode);
511             prec = longDigitLength(rs);
512             drop = prec - mcp;
513         }
514     }
515     } else {
516         char coeff[] = new char[len];
517         for (; len > 0; offset++, len--) {
518             c = in[offset];
519             // have digit
520             if ((c >= '0' && c <= '9') || Character.isDigit(c)) {
521                 // First compact case, we need not to preserve the character
522                 // and we can just compute the value in place.
523                 if (c == '0' || Character.digit(c, 10) == 0) {
524                     if (prec == 0) {
525                         coeff[idx] = c;
526                         prec = 1;
527                     } else if (idx != 0) {
528                         coeff[idx++] = c;
529                         ++prec;
530                     } // else c must be a redundant leading zero
531                 } else {
532                     if (prec != 1 || idx != 0)
533                         ++prec; // prec unchanged if preceded by 0s
534                     coeff[idx++] = c;
535                 }
536                 if (dot)
537                     ++scl;
538                 continue;
539             }
540             // have dot
541             if (c == '.') {
542                 // have dot
543                 if (dot) // two dots
544                     throw new NumberFormatException();
545                 dot = true;
546                 continue;
547             }
548             // exponent expected
549             if ((c != 'e') && (c != 'E'))
550                 throw new NumberFormatException();
551             exp = parseExp(in, offset, len);
552             // Next test is required for backwards compatibility
553             if ((int) exp != exp) // overflow

```

```

554         throw new NumberFormatException();
555         break; // [saves a test]
556     }
557     // here when no characters left
558     if (prec == 0) // no digits found
559         throw new NumberFormatException();
560     // Adjust scale if exp is not zero.
561     if (exp != 0) { // had significant exponent
562         scl = adjustScale(scl, exp);
563     }
564     // Remove leading zeros from precision (digits count)
565     rb = new BigInteger(coeff, isneg ? -1 : 1, prec);
566     rs = compactValFor(rb);
567     int mcp = mc.precision;
568     if (mcp > 0 && (prec > mcp)) {
569         if (rs == INFLATED) {
570             int drop = prec - mcp;
571             while (drop > 0) {
572                 scl = checkScaleNonZero((long) scl - drop);
573                 rb = divideAndRoundByTenPow(rb, drop, mc.roundingMode.oldMode);
574                 rs = compactValFor(rb);
575                 if (rs != INFLATED) {
576                     prec = longDigitLength(rs);
577                     break;
578                 }
579                 prec = bigDigitLength(rb);
580                 drop = prec - mcp;
581             }
582         }
583         if (rs != INFLATED) {
584             int drop = prec - mcp;
585             while (drop > 0) {
586                 scl = checkScaleNonZero((long) scl - drop);
587                 rs = divideAndRound(rs, LONG_TEN_POWERS_TABLE[drop], mc.roundingMode.
                    oldMode);
588                 prec = longDigitLength(rs);
589                 drop = prec - mcp;
590             }
591             rb = null;
592         }
593     }
594 }
595 } catch (ArrayIndexOutOfBoundsException e) {
596     throw new NumberFormatException();
597 } catch (NegativeArraySizeException e) {
598     throw new NumberFormatException();
599 }
600 this.scale = scl;
601 this.precision = prec;
602 this.intCompact = rs;
603 this.intVal = rb;
604 }
605

```

```

606     private int adjustScale(int scl, long exp) {
607         long adjustedScale = scl - exp;
608         if (adjustedScale > Integer.MAX_VALUE || adjustedScale < Integer.MIN_VALUE)
609             throw new NumberFormatException("Scale out of range.");
610         scl = (int) adjustedScale;
611         return scl;
612     }
613
614     /*
615     * parse exponent
616     */
617     private static long parseExp(char[] in, int offset, int len){
618         long exp = 0;
619         offset++;
620         char c = in[offset];
621         len--;
622         boolean negexp = (c == '-');
623         // optional sign
624         if (negexp || c == '+') {
625             offset++;
626             c = in[offset];
627             len--;
628         }
629         if (len <= 0) // no exponent digits
630             throw new NumberFormatException();
631         // skip leading zeros in the exponent
632         while (len > 10 && (c=='0' || (Character.digit(c, 10) == 0))) {
633             offset++;
634             c = in[offset];
635             len--;
636         }
637         if (len > 10) // too many nonzero exponent digits
638             throw new NumberFormatException();
639         // c now holds first digit of exponent
640         for (;;) {
641             int v;
642             if (c >= '0' && c <= '9') {
643                 v = c - '0';
644             } else {
645                 v = Character.digit(c, 10);
646                 if (v < 0) // not a digit
647                     throw new NumberFormatException();
648             }
649             exp = exp * 10 + v;
650             if (len == 1)
651                 break; // that was final character
652             offset++;
653             c = in[offset];
654         }
655         if (negexp) // apply sign
656             exp = -exp;
657         return exp;
658     }

```

```

659
660 /**
661  * Translates a character array representation of a
662  * {@code BigDecimal} into a {@code BigDecimal}, accepting the
663  * same sequence of characters as the {@link #BigDecimal(String)}
664  * constructor.
665  *
666  * <p>Note that if the sequence of characters is already available
667  * as a character array, using this constructor is faster than
668  * converting the {@code char} array to string and using the
669  * {@code BigDecimal(String)} constructor .
670  *
671  * @param in {@code char} array that is the source of characters.
672  * @throws NumberFormatException if {@code in} is not a valid
673  *         representation of a {@code BigDecimal}.
674  * @since 1.5
675  */
676 public BigDecimal(char[] in) {
677     this(in, 0, in.length);
678 }
679
680 /**
681  * Translates a character array representation of a
682  * {@code BigDecimal} into a {@code BigDecimal}, accepting the
683  * same sequence of characters as the {@link #BigDecimal(String)}
684  * constructor and with rounding according to the context
685  * settings.
686  *
687  * <p>Note that if the sequence of characters is already available
688  * as a character array, using this constructor is faster than
689  * converting the {@code char} array to string and using the
690  * {@code BigDecimal(String)} constructor .
691  *
692  * @param in {@code char} array that is the source of characters.
693  * @param mc the context to use.
694  * @throws ArithmeticException if the result is inexact but the
695  *         rounding mode is {@code UNNECESSARY}.
696  * @throws NumberFormatException if {@code in} is not a valid
697  *         representation of a {@code BigDecimal}.
698  * @since 1.5
699  */
700 public BigDecimal(char[] in, MathContext mc) {
701     this(in, 0, in.length, mc);
702 }
703
704 /**
705  * Translates the string representation of a {@code BigDecimal}
706  * into a {@code BigDecimal}. The string representation consists
707  * of an optional sign, {@code '+'} (<tt>'&#92;u002B'</tt>) or
708  * {@code '-'} (<tt>'&#92;u002D'</tt>), followed by a sequence of
709  * zero or more decimal digits ("the integer"), optionally
710  * followed by a fraction, optionally followed by an exponent.
711  *

```

```

712 * <p>The fraction consists of a decimal point followed by zero
713 * or more decimal digits. The string must contain at least one
714 * digit in either the integer or the fraction. The number formed
715 * by the sign, the integer and the fraction is referred to as the
716 * <i>significand</i>.
717 *
718 * <p>The exponent consists of the character {@code 'e'}
719 * (<tt>'&#92;u0065'</tt>) or {@code 'E'} (<tt>'&#92;u0045'</tt>)
720 * followed by one or more decimal digits. The value of the
721 * exponent must lie between -{@link Integer#MAX_VALUE} ({@link
722 * Integer#MIN_VALUE}+1) and {@link Integer#MAX_VALUE}, inclusive.
723 *
724 * <p>More formally, the strings this constructor accepts are
725 * described by the following grammar:
726 * <blockquote>
727 * <dl>
728 * <dt><i>BigDecimalString:</i>
729 * <dd><i>Sign<sub>opt</sub> Significand Exponent<sub>opt</sub></i>
730 * <dt><i>Sign:</i>
731 * <dd>{@code +}
732 * <dd>{@code -}
733 * <dt><i>Significand:</i>
734 * <dd><i>IntegerPart</i> {@code .} <i>FractionPart<sub>opt</sub></i>
735 * <dd>{@code .} <i>FractionPart</i>
736 * <dd><i>IntegerPart</i>
737 * <dt><i>IntegerPart:</i>
738 * <dd><i>Digits</i>
739 * <dt><i>FractionPart:</i>
740 * <dd><i>Digits</i>
741 * <dt><i>Exponent:</i>
742 * <dd><i>ExponentIndicator SignedInteger</i>
743 * <dt><i>ExponentIndicator:</i>
744 * <dd>{@code e}
745 * <dd>{@code E}
746 * <dt><i>SignedInteger:</i>
747 * <dd><i>Sign<sub>opt</sub> Digits</i>
748 * <dt><i>Digits:</i>
749 * <dd><i>Digit</i>
750 * <dd><i>Digits Digit</i>
751 * <dt><i>Digit:</i>
752 * <dd>any character for which {@link Character#isDigit}
753 * returns {@code true}, including 0, 1, 2 ...
754 * </dl>
755 * </blockquote>
756 *
757 * <p>The scale of the returned {@code BigDecimal} will be the
758 * number of digits in the fraction, or zero if the string
759 * contains no decimal point, subject to adjustment for any
760 * exponent; if the string contains an exponent, the exponent is
761 * subtracted from the scale. The value of the resulting scale
762 * must lie between {@code Integer.MIN_VALUE} and
763 * {@code Integer.MAX_VALUE}, inclusive.
764 *

```

```

765 * <p>The character-to-digit mapping is provided by {@link
766 * java.lang.Character#digit} set to convert to radix 10. The
767 * String may not contain any extraneous characters (whitespace,
768 * for example).
769 *
770 * <p><b>Examples:</b><br>
771 * The value of the returned {@code BigDecimal} is equal to
772 * <i>significand</i> <i>times</i> 10<sup>&nbsp;<i>exponent</i></sup>.
773 * For each string on the left, the resulting representation
774 * [{@code BigInteger}, {@code scale}] is shown on the right.
775 * <pre>
776 * "0" [0,0]
777 * "0.00" [0,2]
778 * "123" [123,0]
779 * "-123" [-123,0]
780 * "1.23E3" [123,-1]
781 * "1.23E+3" [123,-1]
782 * "12.3E+7" [123,-6]
783 * "12.0" [120,1]
784 * "12.3" [123,1]
785 * "0.00123" [123,5]
786 * "-1.23E-12" [-123,14]
787 * "1234.5E-4" [12345,5]
788 * "0E+7" [0,-7]
789 * "-0" [0,0]
790 * </pre>
791 *
792 * <p>Note: For values other than {@code float} and
793 * {@code double} NaN and &plusmn;Infinity, this constructor is
794 * compatible with the values returned by {@link Float#toString}
795 * and {@link Double#toString}. This is generally the preferred
796 * way to convert a {@code float} or {@code double} into a
797 * BigDecimal, as it doesn't suffer from the unpredictability of
798 * the {@link #BigDecimal\(double\)} constructor.
799 *
800 * @param val String representation of {@code BigDecimal}.
801 *
802 * @throws NumberFormatException if {@code val} is not a valid
803 * representation of a {@code BigDecimal}.
804 */
805 public BigDecimal(String val) {
806     this(val.toCharArray(), 0, val.length());
807 }
808
809 /**
810 * Translates the string representation of a {@code BigDecimal}
811 * into a {@code BigDecimal}, accepting the same strings as the
812 * {@link #BigDecimal\(String\)} constructor, with rounding
813 * according to the context settings.
814 *
815 * @param val string representation of a {@code BigDecimal}.
816 * @param mc the context to use.
817 * @throws ArithmeticException if the result is inexact but the

```

```

818 *      rounding mode is {@code UNNECESSARY}.
819 * @throws NumberFormatException if {@code val} is not a valid
820 *      representation of a BigDecimal.
821 * @since 1.5
822 */
823 public BigDecimal(String val, MathContext mc) {
824     this(val.toCharArray(), 0, val.length(), mc);
825 }
826
827 /**
828 * Translates a {@code double} into a {@code BigDecimal} which
829 * is the exact decimal representation of the {@code double}'s
830 * binary floating-point value. The scale of the returned
831 * {@code BigDecimal} is the smallest value such that
832 *  $10^{\text{scale}} \times \text{val}$  is an integer.
833 * <p>
834 * <b>Notes:</b>
835 * <ol>
836 * <li>
837 * The results of this constructor can be somewhat unpredictable.
838 * One might assume that writing {@code new BigDecimal(0.1)} in
839 * Java creates a {@code BigDecimal} which is exactly equal to
840 * 0.1 (an unscaled value of 1, with a scale of 1), but it is
841 * actually equal to
842 * 0.1000000000000000055511151231257827021181583404541015625.
843 * This is because 0.1 cannot be represented exactly as a
844 * {@code double} (or, for that matter, as a binary fraction of
845 * any finite length). Thus, the value that is being passed
846 * <i>in</i> to the constructor is not exactly equal to 0.1,
847 * appearances notwithstanding.
848 *
849 * <li>
850 * The {@code String} constructor, on the other hand, is
851 * perfectly predictable: writing {@code new BigDecimal("0.1")}
852 * creates a {@code BigDecimal} which is <i>exactly</i> equal to
853 * 0.1, as one would expect. Therefore, it is generally
854 * recommended that the {@link #BigDecimal(String)}
855 * String constructor be used in preference to this one.
856 *
857 * <li>
858 * When a {@code double} must be used as a source for a
859 * {@code BigDecimal}, note that this constructor provides an
860 * exact conversion; it does not give the same result as
861 * converting the {@code double} to a {@code String} using the
862 * {@link Double#toString(double)} method and then using the
863 * {@link #BigDecimal(String)} constructor. To get that result,
864 * use the {@code static} {@link #valueOf(double)} method.
865 * </ol>
866 *
867 * @param val {@code double} value to be converted to
868 *      {@code BigDecimal}.
869 * @throws NumberFormatException if {@code val} is infinite or NaN.
870 */

```



```

871 public BigDecimal(double val) {
872     this(val, MathContext.UNLIMITED);
873 }
874
875 /**
876  * Translates a {@code double} into a {@code BigDecimal}, with
877  * rounding according to the context settings. The scale of the
878  * {@code BigDecimal} is the smallest value such that
879  *  $\langle 10^{\text{scale}} \times \text{val} \rangle$  is an integer.
880  *
881  * <p>The results of this constructor can be somewhat unpredictable
882  * and its use is generally not recommended; see the notes under
883  * the {@link #BigDecimal(double)} constructor.
884  *
885  * @param val {@code double} value to be converted to
886  *             {@code BigDecimal}.
887  * @param mc the context to use.
888  * @throws ArithmeticException if the result is inexact but the
889  *             RoundingMode is UNNECESSARY.
890  * @throws NumberFormatException if {@code val} is infinite or NaN.
891  * @since 1.5
892  */
893 public BigDecimal(double val, MathContext mc) {
894     if (Double.isInfinite(val) || Double.isNaN(val))
895         throw new NumberFormatException("Infinite or NaN");
896     // Translate the double into sign, exponent and significand, according
897     // to the formulae in JLS, Section 20.10.22.
898     long valBits = Double.doubleToLongBits(val);
899     int sign = ((valBits >> 63) == 0 ? 1 : -1);
900     int exponent = (int) ((valBits >> 52) & 0x7ffL);
901     long significand = (exponent == 0
902         ? (valBits & ((1L << 52) - 1)) << 1
903         : (valBits & ((1L << 52) - 1)) | (1L << 52));
904     exponent -= 1075;
905     // At this point, val == sign * significand * 2**exponent.
906
907     /*
908     * Special case zero to suppress nonterminating normalization and bogus
909     * scale calculation.
910     */
911     if (significand == 0) {
912         this.intVal = BigInteger.ZERO;
913         this.scale = 0;
914         this.intCompact = 0;
915         this.precision = 1;
916         return;
917     }
918     // Normalize
919     while ((significand & 1) == 0) { // i.e., significand is even
920         significand >>= 1;
921         exponent++;
922     }
923     int scale = 0;

```

```

924     // Calculate intVal and scale
925     BigInteger intVal;
926     long compactVal = sign * significand;
927     if (exponent == 0) {
928         intVal = (compactVal == INFLATED) ? INFLATED_BIGINT : null;
929     } else {
930         if (exponent < 0) {
931             intVal = BigInteger.valueOf(5).pow(-exponent).multiply(compactVal);
932             scale = -exponent;
933         } else { // (exponent > 0)
934             intVal = BigInteger.valueOf(2).pow(exponent).multiply(compactVal);
935         }
936         compactVal = compactValFor(intVal);
937     }
938     int prec = 0;
939     int mcp = mc.precision;
940     if (mcp > 0) { // do rounding
941         int mode = mc.roundingMode.oldMode;
942         int drop;
943         if (compactVal == INFLATED) {
944             prec = bigDigitLength(intVal);
945             drop = prec - mcp;
946             while (drop > 0) {
947                 scale = checkScaleNonZero((long) scale - drop);
948                 intVal = divideAndRoundByTenPow(intVal, drop, mode);
949                 compactVal = compactValFor(intVal);
950                 if (compactVal != INFLATED) {
951                     break;
952                 }
953                 prec = bigDigitLength(intVal);
954                 drop = prec - mcp;
955             }
956         }
957         if (compactVal != INFLATED) {
958             prec = longDigitLength(compactVal);
959             drop = prec - mcp;
960             while (drop > 0) {
961                 scale = checkScaleNonZero((long) scale - drop);
962                 compactVal = divideAndRound(compactVal, LONG_TEN_POWERS_TABLE[drop], mc.
                    roundingMode.oldMode);
963                 prec = longDigitLength(compactVal);
964                 drop = prec - mcp;
965             }
966             intVal = null;
967         }
968     }
969     this.intVal = intVal;
970     this.intCompact = compactVal;
971     this.scale = scale;
972     this.precision = prec;
973 }
974
975 /**

```

```

976     * Translates a {@code BigInteger} into a {@code BigDecimal}.
977     * The scale of the {@code BigDecimal} is zero.
978     *
979     * @param val {@code BigInteger} value to be converted to
980     *           {@code BigDecimal}.
981     */
982     public BigDecimal(BigInteger val) {
983         scale = 0;
984         intVal = val;
985         intCompact = compactValFor(val);
986     }
987
988     /**
989     * Translates a {@code BigInteger} into a {@code BigDecimal}
990     * rounding according to the context settings. The scale of the
991     * {@code BigDecimal} is zero.
992     *
993     * @param val {@code BigInteger} value to be converted to
994     *           {@code BigDecimal}.
995     * @param mc the context to use.
996     * @throws ArithmeticException if the result is inexact but the
997     *         rounding mode is {@code UNNECESSARY}.
998     * @since 1.5
999     */
1000    public BigDecimal(BigInteger val, MathContext mc) {
1001        this(val,0,mc);
1002    }
1003
1004    /**
1005     * Translates a {@code BigInteger} unscaled value and an
1006     * {@code int} scale into a {@code BigDecimal}. The value of
1007     * the {@code BigDecimal} is
1008     * <tt>(unscaledVal  $\times$  10-scale)</tt>.
1009     *
1010     * @param unscaledVal unscaled value of the {@code BigDecimal}.
1011     * @param scale scale of the {@code BigDecimal}.
1012     */
1013    public BigDecimal(BigInteger unscaledVal, int scale) {
1014        // Negative scales are now allowed
1015        this.intVal = unscaledVal;
1016        this.intCompact = compactValFor(unscaledVal);
1017        this.scale = scale;
1018    }
1019
1020    /**
1021     * Translates a {@code BigInteger} unscaled value and an
1022     * {@code int} scale into a {@code BigDecimal}, with rounding
1023     * according to the context settings. The value of the
1024     * {@code BigDecimal} is <tt>(unscaledVal  $\times$ 
1025     * 10-scale)</tt>, rounded according to the
1026     * {@code precision} and rounding mode settings.
1027     *
1028     * @param unscaledVal unscaled value of the {@code BigDecimal}.

```

```

1029     * @param scale scale of the {@code BigDecimal}.
1030     * @param mc the context to use.
1031     * @throws ArithmeticException if the result is inexact but the
1032     *         rounding mode is {@code UNNECESSARY}.
1033     * @since 1.5
1034     */
1035     public BigDecimal(BigInteger unscaledVal, int scale, MathContext mc) {
1036         long compactVal = compactValFor(unscaledVal);
1037         int mcp = mc.precision;
1038         int prec = 0;
1039         if (mcp > 0) { // do rounding
1040             int mode = mc.roundingMode.oldMode;
1041             if (compactVal == INFLATED) {
1042                 prec = bigDigitLength(unscaledVal);
1043                 int drop = prec - mcp;
1044                 while (drop > 0) {
1045                     scale = checkScaleNonZero((long) scale - drop);
1046                     unscaledVal = divideAndRoundByTenPow(unscaledVal, drop, mode);
1047                     compactVal = compactValFor(unscaledVal);
1048                     if (compactVal != INFLATED) {
1049                         break;
1050                     }
1051                     prec = bigDigitLength(unscaledVal);
1052                     drop = prec - mcp;
1053                 }
1054             }
1055             if (compactVal != INFLATED) {
1056                 prec = longDigitLength(compactVal);
1057                 int drop = prec - mcp; // drop can't be more than 18
1058                 while (drop > 0) {
1059                     scale = checkScaleNonZero((long) scale - drop);
1060                     compactVal = divideAndRound(compactVal, LONG_TEN_POWERS_TABLE[drop], mode);
1061                     prec = longDigitLength(compactVal);
1062                     drop = prec - mcp;
1063                 }
1064                 unscaledVal = null;
1065             }
1066         }
1067         this.intVal = unscaledVal;
1068         this.intCompact = compactVal;
1069         this.scale = scale;
1070         this.precision = prec;
1071     }
1072
1073     /**
1074     * Translates an {@code int} into a {@code BigDecimal}. The
1075     * scale of the {@code BigDecimal} is zero.
1076     *
1077     * @param val {@code int} value to be converted to
1078     *             {@code BigDecimal}.
1079     * @since 1.5
1080     */
1081     public BigDecimal(int val) {

```

```

1082     this.intCompact = val;
1083     this.scale = 0;
1084     this.intVal = null;
1085 }
1086
1087 /**
1088  * Translates an {@code int} into a {@code BigDecimal}, with
1089  * rounding according to the context settings. The scale of the
1090  * {@code BigDecimal}, before any rounding, is zero.
1091  *
1092  * @param val {@code int} value to be converted to {@code BigDecimal}.
1093  * @param mc the context to use.
1094  * @throws ArithmeticException if the result is inexact but the
1095  *         rounding mode is {@code UNNECESSARY}.
1096  * @since 1.5
1097  */
1098 public BigDecimal(int val, MathContext mc) {
1099     int mcp = mc.precision;
1100     long compactVal = val;
1101     int scale = 0;
1102     int prec = 0;
1103     if (mcp > 0) { // do rounding
1104         prec = longDigitLength(compactVal);
1105         int drop = prec - mcp; // drop can't be more than 18
1106         while (drop > 0) {
1107             scale = checkScaleNonZero((long) scale - drop);
1108             compactVal = divideAndRound(compactVal, LONG_TEN_POWERS_TABLE[drop], mc,
1109                                     roundingMode.oldMode);
1109             prec = longDigitLength(compactVal);
1110             drop = prec - mcp;
1111         }
1112     }
1113     this.intVal = null;
1114     this.intCompact = compactVal;
1115     this.scale = scale;
1116     this.precision = prec;
1117 }
1118
1119 /**
1120  * Translates a {@code long} into a {@code BigDecimal}. The
1121  * scale of the {@code BigDecimal} is zero.
1122  *
1123  * @param val {@code long} value to be converted to {@code BigDecimal}.
1124  * @since 1.5
1125  */
1126 public BigDecimal(long val) {
1127     this.intCompact = val;
1128     this.intVal = (val == INFLATED) ? INFLATED_BIGINT : null;
1129     this.scale = 0;
1130 }
1131
1132 /**
1133  * Translates a {@code long} into a {@code BigDecimal}, with

```

```

1134     * rounding according to the context settings. The scale of the
1135     * {@code BigDecimal}, before any rounding, is zero.
1136     *
1137     * @param val {@code long} value to be converted to {@code BigDecimal}.
1138     * @param mc the context to use.
1139     * @throws ArithmeticException if the result is inexact but the
1140     *         rounding mode is {@code UNNECESSARY}.
1141     * @since 1.5
1142     */
1143     public BigDecimal(long val, MathContext mc) {
1144         int mcp = mc.precision;
1145         int mode = mc.roundingMode.oldMode;
1146         int prec = 0;
1147         int scale = 0;
1148         BigInteger intVal = (val == INFLATED) ? INFLATED_BIGINT : null;
1149         if (mcp > 0) { // do rounding
1150             if (val == INFLATED) {
1151                 prec = 19;
1152                 int drop = prec - mcp;
1153                 while (drop > 0) {
1154                     scale = checkScaleNonZero((long) scale - drop);
1155                     intVal = divideAndRoundByTenPow(intVal, drop, mode);
1156                     val = compactValFor(intVal);
1157                     if (val != INFLATED) {
1158                         break;
1159                     }
1160                     prec = bigDigitLength(intVal);
1161                     drop = prec - mcp;
1162                 }
1163             }
1164             if (val != INFLATED) {
1165                 prec = longDigitLength(val);
1166                 int drop = prec - mcp;
1167                 while (drop > 0) {
1168                     scale = checkScaleNonZero((long) scale - drop);
1169                     val = divideAndRound(val, LONG_TEN_POWERS_TABLE[drop], mc.roundingMode.oldMode);
1170                     ;
1171                     prec = longDigitLength(val);
1172                     drop = prec - mcp;
1173                 }
1174                 intVal = null;
1175             }
1176             this.intVal = intVal;
1177             this.intCompact = val;
1178             this.scale = scale;
1179             this.precision = prec;
1180         }
1181
1182         // Static Factory Methods
1183
1184         /**
1185          * Translates a {@code long} unscaled value and an

```

```

1186 * {@code int} scale into a {@code BigDecimal}. This
1187 * {@literal "static factory method"} is provided in preference to
1188 * a ({@code long}, {@code int}) constructor because it
1189 * allows for reuse of frequently used {@code BigDecimal} values..
1190 *
1191 * @param unscaledVal unscaled value of the {@code BigDecimal}.
1192 * @param scale scale of the {@code BigDecimal}.
1193 * @return a {@code BigDecimal} whose value is
1194 *         
$$\text{unscaledVal} \times 10^{\text{scale}}$$

1195 */
1196 public static BigDecimal valueOf(long unscaledVal, int scale) {
1197     if (scale == 0)
1198         return valueOf(unscaledVal);
1199     else if (unscaledVal == 0) {
1200         return zeroValueOf(scale);
1201     }
1202     return new BigDecimal(unscaledVal == INFLATED ?
1203                           INFLATED_BIGINT : null,
1204                           unscaledVal, scale, 0);
1205 }
1206
1207 /**
1208  * Translates a {@code long} value into a {@code BigDecimal}
1209  * with a scale of zero. This {@literal "static factory method"}
1210  * is provided in preference to a ({@code long}) constructor
1211  * because it allows for reuse of frequently used
1212  * {@code BigDecimal} values.
1213  *
1214  * @param val value of the {@code BigDecimal}.
1215  * @return a {@code BigDecimal} whose value is {@code val}.
1216  */
1217 public static BigDecimal valueOf(long val) {
1218     if (val >= 0 && val < zeroThroughTen.length)
1219         return zeroThroughTen[(int)val];
1220     else if (val != INFLATED)
1221         return new BigDecimal(null, val, 0, 0);
1222     return new BigDecimal(INFLATED_BIGINT, val, 0, 0);
1223 }
1224
1225 static BigDecimal valueOf(long unscaledVal, int scale, int prec) {
1226     if (scale == 0 && unscaledVal >= 0 && unscaledVal < zeroThroughTen.length) {
1227         return zeroThroughTen[(int) unscaledVal];
1228     } else if (unscaledVal == 0) {
1229         return zeroValueOf(scale);
1230     }
1231     return new BigDecimal(unscaledVal == INFLATED ? INFLATED_BIGINT : null,
1232                           unscaledVal, scale, prec);
1233 }
1234
1235 static BigDecimal valueOf(BigInteger intVal, int scale, int prec) {
1236     long val = compactValFor(intVal);
1237     if (val == 0) {
1238         return zeroValueOf(scale);

```

```

1239     } else if (scale == 0 && val >= 0 && val < zeroThroughTen.length) {
1240         return zeroThroughTen[(int) val];
1241     }
1242     return new BigDecimal(intVal, val, scale, prec);
1243 }
1244
1245 static BigDecimal zeroValueOf(int scale) {
1246     if (scale >= 0 && scale < ZERO_SCALED_BY.length)
1247         return ZERO_SCALED_BY[scale];
1248     else
1249         return new BigDecimal(BigInteger.ZERO, 0, scale, 1);
1250 }
1251
1252 /**
1253  * Translates a {@code double} into a {@code BigDecimal}, using
1254  * the {@code double}'s canonical string representation provided
1255  * by the {@link Double#toString(double)} method.
1256  *
1257  * <p><b>Note:</b> This is generally the preferred way to convert
1258  * a {@code double} (or {@code float}) into a
1259  * {@code BigDecimal}, as the value returned is equal to that
1260  * resulting from constructing a {@code BigDecimal} from the
1261  * result of using {@link Double#toString(double)}.
1262  *
1263  * @param val {@code double} to convert to a {@code BigDecimal}.
1264  * @return a {@code BigDecimal} whose value is equal to or approximately
1265  *         equal to the value of {@code val}.
1266  * @throws NumberFormatException if {@code val} is infinite or NaN.
1267  * @since 1.5
1268  */
1269 public static BigDecimal valueOf(double val) {
1270     // Reminder: a zero double returns '0.0', so we cannot fastpath
1271     // to use the constant ZERO. This might be important enough to
1272     // justify a factory approach, a cache, or a few private
1273     // constants, later.
1274     return new BigDecimal(Double.toString(val));
1275 }
1276
1277 // Arithmetic Operations
1278 /**
1279  * Returns a {@code BigDecimal} whose value is {@code (this +
1280  * augend)}, and whose scale is {@code max(this.scale(),
1281  * augend.scale())}.
1282  *
1283  * @param augend value to be added to this {@code BigDecimal}.
1284  * @return {@code this + augend}
1285  */
1286 public BigDecimal add(BigDecimal augend) {
1287     if (this.intCompact != INFLATED) {
1288         if ((augend.intCompact != INFLATED)) {
1289             return add(this.intCompact, this.scale, augend.intCompact, augend.scale);
1290         } else {
1291             return add(this.intCompact, this.scale, augend.intVal, augend.scale);

```



```

1292     }
1293     } else {
1294         if ((augend.intCompact != INFLATED)) {
1295             return add(augend.intCompact, augend.scale, this.intVal, this.scale);
1296         } else {
1297             return add(this.intVal, this.scale, augend.intVal, augend.scale);
1298         }
1299     }
1300 }
1301
1302 /**
1303  * Returns a {@code BigDecimal} whose value is {@code (this + augend)},
1304  * with rounding according to the context settings.
1305  *
1306  * If either number is zero and the precision setting is nonzero then
1307  * the other number, rounded if necessary, is used as the result.
1308  *
1309  * @param augend value to be added to this {@code BigDecimal}.
1310  * @param mc the context to use.
1311  * @return {@code this + augend}, rounded as necessary.
1312  * @throws ArithmeticException if the result is inexact but the
1313  *         rounding mode is {@code UNNECESSARY}.
1314  * @since 1.5
1315  */
1316 public BigDecimal add(BigDecimal augend, MathContext mc) {
1317     if (mc.precision == 0)
1318         return add(augend);
1319     BigDecimal lhs = this;
1320
1321     // If either number is zero then the other number, rounded and
1322     // scaled if necessary, is used as the result.
1323     {
1324         boolean lhsIsZero = lhs.signum() == 0;
1325         boolean augendIsZero = augend.signum() == 0;
1326
1327         if (lhsIsZero || augendIsZero) {
1328             int preferredScale = Math.max(lhs.scale(), augend.scale());
1329             BigDecimal result;
1330
1331             if (lhsIsZero && augendIsZero)
1332                 return zeroValueOf(preferredScale);
1333             result = lhsIsZero ? doRound(augend, mc) : doRound(lhs, mc);
1334
1335             if (result.scale() == preferredScale)
1336                 return result;
1337             else if (result.scale() > preferredScale) {
1338                 return stripZerosToMatchScale(result.intVal, result.intCompact, result.scale,
1339                     preferredScale);
1340             } else { // result.scale < preferredScale
1341                 int precisionDiff = mc.precision - result.precision();
1342                 int scaleDiff = preferredScale - result.scale();
1343
1344                 if (precisionDiff >= scaleDiff)

```

```

1344         return result.setScale(preferredScale); // can achieve target scale
1345     else
1346         return result.setScale(result.scale() + precisionDiff);
1347     }
1348 }
1349 }
1350
1351 long padding = (long) lhs.scale - augend.scale;
1352 if (padding != 0) { // scales differ; alignment needed
1353     BigDecimal arg[] = preAlign(lhs, augend, padding, mc);
1354     matchScale(arg);
1355     lhs = arg[0];
1356     augend = arg[1];
1357 }
1358 return doRound(lhs.inflated().add(augend.inflated()), lhs.scale, mc);
1359 }
1360
1361 /**
1362  * Returns an array of length two, the sum of whose entries is
1363  * equal to the rounded sum of the {@code BigDecimal} arguments.
1364  *
1365  * <p>If the digit positions of the arguments have a sufficient
1366  * gap between them, the value smaller in magnitude can be
1367  * condensed into a {@literal "sticky bit"} and the end result will
1368  * round the same way <em>if</em> the precision of the final
1369  * result does not include the high order digit of the small
1370  * magnitude operand.
1371  *
1372  * <p>Note that while strictly speaking this is an optimization,
1373  * it makes a much wider range of additions practical.
1374  *
1375  * <p>This corresponds to a pre-shift operation in a fixed
1376  * precision floating-point adder; this method is complicated by
1377  * variable precision of the result as determined by the
1378  * MathContext. A more nuanced operation could implement a
1379  * {@literal "right shift"} on the smaller magnitude operand so
1380  * that the number of digits of the smaller operand could be
1381  * reduced even though the significands partially overlapped.
1382  */
1383 private BigDecimal[] preAlign(BigDecimal lhs, BigDecimal augend, long padding, MathContext mc)
1384 {
1385     assert padding != 0;
1386     BigDecimal big;
1387     BigDecimal small;
1388
1389     if (padding < 0) { // lhs is big; augend is small
1390         big = lhs;
1391         small = augend;
1392     } else { // lhs is small; augend is big
1393         big = augend;
1394         small = lhs;
1395     }

```

```

1396     /*
1397      * This is the estimated scale of an ulp of the result; it assumes that
1398      * the result doesn't have a carry-out on a true add (e.g. 999 + 1 =>
1399      * 1000) or any subtractive cancellation on borrowing (e.g. 100 - 1.2 =>
1400      * 98.8)
1401      */
1402     long estResultUlpScale = (long) big.scale - big.precision() + mc.precision;
1403
1404     /*
1405      * The low-order digit position of big is big.scale(). This
1406      * is true regardless of whether big has a positive or
1407      * negative scale. The high-order digit position of small is
1408      * small.scale - (small.precision() - 1). To do the full
1409      * condensation, the digit positions of big and small must be
1410      * disjoint *and* the digit positions of small should not be
1411      * directly visible in the result.
1412      */
1413     long smallHighDigitPos = (long) small.scale - small.precision() + 1;
1414     if (smallHighDigitPos > big.scale + 2 && // big and small disjoint
1415         smallHighDigitPos > estResultUlpScale + 2) { // small digits not visible
1416         small = BigDecimal.valueOf(small.signum(), this.checkScale(Math.max(big.scale,
1417             estResultUlpScale) + 3));
1418     }
1419
1420     // Since addition is symmetric, preserving input order in
1421     // returned operands doesn't matter
1422     BigDecimal[] result = {big, small};
1423     return result;
1424 }
1425
1426 /**
1427  * Returns a {@code BigDecimal} whose value is {@code (this -
1428  * subtrahend)}, and whose scale is {@code max(this.scale(),
1429  * subtrahend.scale())}.
1430  *
1431  * @param subtrahend value to be subtracted from this {@code BigDecimal}.
1432  * @return {@code this - subtrahend}
1433  */
1434 public BigDecimal subtract(BigDecimal subtrahend) {
1435     if (this.intCompact != INFLATED) {
1436         if ((subtrahend.intCompact != INFLATED)) {
1437             return add(this.intCompact, this.scale, -subtrahend.intCompact, subtrahend.scale);
1438         } else {
1439             return add(this.intCompact, this.scale, subtrahend.intVal.negate(), subtrahend.
1440                 scale);
1441         }
1442     } else {
1443         if ((subtrahend.intCompact != INFLATED)) {
1444             // Pair of subtrahend values given before pair of
1445             // values from this BigDecimal to avoid need for
1446             // method overloading on the specialized add method
1447             return add(-subtrahend.intCompact, subtrahend.scale, this.intVal, this.scale);
1448         } else {

```

```

1447         return add(this.intVal, this.scale, subtrahend.intVal.negate(), subtrahend.scale);
1448     }
1449 }
1450 }
1451
1452 /**
1453  * Returns a {@code BigDecimal} whose value is {@code (this - subtrahend)},
1454  * with rounding according to the context settings.
1455  *
1456  * If {@code subtrahend} is zero then this, rounded if necessary, is used as the
1457  * result. If this is zero then the result is {@code subtrahend.negate(mc)}.
1458  *
1459  * @param subtrahend value to be subtracted from this {@code BigDecimal}.
1460  * @param mc the context to use.
1461  * @return {@code this - subtrahend}, rounded as necessary.
1462  * @throws ArithmeticException if the result is inexact but the
1463  *         rounding mode is {@code UNNECESSARY}.
1464  * @since 1.5
1465  */
1466 public BigDecimal subtract(BigDecimal subtrahend, MathContext mc) {
1467     if (mc.precision == 0)
1468         return subtract(subtrahend);
1469     // share the special rounding code in add()
1470     return add(subtrahend.negate(), mc);
1471 }
1472
1473 /**
1474  * Returns a {@code BigDecimal} whose value is <tt>(this &times;
1475  * multiplicand)</tt>, and whose scale is {@code (this.scale() +
1476  * multiplicand.scale())}.
1477  *
1478  * @param multiplicand value to be multiplied by this {@code BigDecimal}.
1479  * @return {@code this * multiplicand}
1480  */
1481 public BigDecimal multiply(BigDecimal multiplicand) {
1482     int productScale = checkScale((long) scale + multiplicand.scale);
1483     if (this.intCompact != INFLATED) {
1484         if ((multiplicand.intCompact != INFLATED)) {
1485             return multiply(this.intCompact, multiplicand.intCompact, productScale);
1486         } else {
1487             return multiply(this.intCompact, multiplicand.intVal, productScale);
1488         }
1489     } else {
1490         if ((multiplicand.intCompact != INFLATED)) {
1491             return multiply(multiplicand.intCompact, this.intVal, productScale);
1492         } else {
1493             return multiply(this.intVal, multiplicand.intVal, productScale);
1494         }
1495     }
1496 }
1497
1498 /**
1499  * Returns a {@code BigDecimal} whose value is <tt>(this &times;

```

```

1500     * multiplicand)</tt>, with rounding according to the context settings.
1501     *
1502     * @param multiplicand value to be multiplied by this {@code BigDecimal}.
1503     * @param mc the context to use.
1504     * @return {@code this * multiplicand}, rounded as necessary.
1505     * @throws ArithmeticException if the result is inexact but the
1506     *         rounding mode is {@code UNNECESSARY}.
1507     * @since 1.5
1508     */
1509     public BigDecimal multiply(BigDecimal multiplicand, MathContext mc) {
1510         if (mc.precision == 0)
1511             return multiply(multiplicand);
1512         int productScale = checkScale((long) scale + multiplicand.scale);
1513         if (this.intCompact != INFLATED) {
1514             if ((multiplicand.intCompact != INFLATED)) {
1515                 return multiplyAndRound(this.intCompact, multiplicand.intCompact, productScale, mc)
1516                     ;
1517             } else {
1518                 return multiplyAndRound(this.intCompact, multiplicand.intVal, productScale, mc);
1519             }
1520         } else {
1521             if ((multiplicand.intCompact != INFLATED)) {
1522                 return multiplyAndRound(multiplicand.intCompact, this.intVal, productScale, mc);
1523             } else {
1524                 return multiplyAndRound(this.intVal, multiplicand.intVal, productScale, mc);
1525             }
1526         }
1527     }
1528     /**
1529     * Returns a {@code BigDecimal} whose value is {@code (this /
1530     * divisor)}, and whose scale is as specified. If rounding must
1531     * be performed to generate a result with the specified scale, the
1532     * specified rounding mode is applied.
1533     *
1534     * <p>The new {@link #divide(BigDecimal, int, RoundingMode)} method
1535     * should be used in preference to this legacy method.
1536     *
1537     * @param divisor value by which this {@code BigDecimal} is to be divided.
1538     * @param scale scale of the {@code BigDecimal} quotient to be returned.
1539     * @param roundingMode rounding mode to apply.
1540     * @return {@code this / divisor}
1541     * @throws ArithmeticException if {@code divisor} is zero,
1542     *         {@code roundingMode==ROUND_UNNECESSARY} and
1543     *         the specified scale is insufficient to represent the result
1544     *         of the division exactly.
1545     * @throws IllegalArgumentException if {@code roundingMode} does not
1546     *         represent a valid rounding mode.
1547     * @see #ROUND_UP
1548     * @see #ROUND_DOWN
1549     * @see #ROUND_CEILING
1550     * @see #ROUND_FLOOR
1551     * @see #ROUND_HALF_UP

```

```

1552     * @see    #ROUND_HALF_DOWN
1553     * @see    #ROUND_HALF_EVEN
1554     * @see    #ROUND_UNNECESSARY
1555     */
1556     public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode) {
1557         if (roundingMode < ROUND_UP || roundingMode > ROUND_UNNECESSARY)
1558             throw new IllegalArgumentException("Invalid rounding mode");
1559         if (this.intCompact != INFLATED) {
1560             if ((divisor.intCompact != INFLATED)) {
1561                 return divide(this.intCompact, this.scale, divisor.intCompact, divisor.scale, scale
1562                     , roundingMode);
1563             } else {
1564                 return divide(this.intCompact, this.scale, divisor.intVal, divisor.scale, scale,
1565                     roundingMode);
1566             }
1567         } else {
1568             if ((divisor.intCompact != INFLATED)) {
1569                 return divide(this.intVal, this.scale, divisor.intCompact, divisor.scale, scale,
1570                     roundingMode);
1571             } else {
1572                 return divide(this.intVal, this.scale, divisor.intVal, divisor.scale, scale,
1573                     roundingMode);
1574             }
1575         }
1576     }
1577
1578     /**
1579     * Returns a {@code BigDecimal} whose value is {@code (this /
1580     * divisor)}, and whose scale is as specified. If rounding must
1581     * be performed to generate a result with the specified scale, the
1582     * specified rounding mode is applied.
1583     *
1584     * @param divisor value by which this {@code BigDecimal} is to be divided.
1585     * @param scale scale of the {@code BigDecimal} quotient to be returned.
1586     * @param roundingMode rounding mode to apply.
1587     * @return {@code this / divisor}
1588     * @throws ArithmeticException if {@code divisor} is zero,
1589     *         {@code roundingMode==RoundingMode.UNNECESSARY} and
1590     *         the specified scale is insufficient to represent the result
1591     *         of the division exactly.
1592     * @since 1.5
1593     */
1594     public BigDecimal divide(BigDecimal divisor, int scale, RoundingMode roundingMode) {
1595         return divide(divisor, scale, roundingMode.oldMode);
1596     }
1597
1598     /**
1599     * Returns a {@code BigDecimal} whose value is {@code (this /
1600     * divisor)}, and whose scale is {@code this.scale()}. If
1601     * rounding must be performed to generate a result with the given
1602     * scale, the specified rounding mode is applied.
1603     *
1604     * <p>The new {@link #divide(BigDecimal, RoundingMode)} method

```

```

1601     * should be used in preference to this legacy method.
1602     *
1603     * @param divisor value by which this {@code BigDecimal} is to be divided.
1604     * @param roundingMode rounding mode to apply.
1605     * @return {@code this / divisor}
1606     * @throws ArithmeticException if {@code divisor==0}, or
1607     *         {@code roundingMode==ROUND_UNNECESSARY} and
1608     *         {@code this.scale()} is insufficient to represent the result
1609     *         of the division exactly.
1610     * @throws IllegalArgumentException if {@code roundingMode} does not
1611     *         represent a valid rounding mode.
1612     * @see     #ROUND_UP
1613     * @see     #ROUND_DOWN
1614     * @see     #ROUND_CEILING
1615     * @see     #ROUND_FLOOR
1616     * @see     #ROUND_HALF_UP
1617     * @see     #ROUND_HALF_DOWN
1618     * @see     #ROUND_HALF_EVEN
1619     * @see     #ROUND_UNNECESSARY
1620     */
1621     public BigDecimal divide(BigDecimal divisor, int roundingMode) {
1622         return this.divide(divisor, scale, roundingMode);
1623     }
1624
1625     /**
1626     * Returns a {@code BigDecimal} whose value is {@code (this /
1627     * divisor)}, and whose scale is {@code this.scale()}. If
1628     * rounding must be performed to generate a result with the given
1629     * scale, the specified rounding mode is applied.
1630     *
1631     * @param divisor value by which this {@code BigDecimal} is to be divided.
1632     * @param roundingMode rounding mode to apply.
1633     * @return {@code this / divisor}
1634     * @throws ArithmeticException if {@code divisor==0}, or
1635     *         {@code roundingMode==RoundingMode.UNNECESSARY} and
1636     *         {@code this.scale()} is insufficient to represent the result
1637     *         of the division exactly.
1638     * @since 1.5
1639     */
1640     public BigDecimal divide(BigDecimal divisor, RoundingMode roundingMode) {
1641         return this.divide(divisor, scale, roundingMode.oldMode);
1642     }
1643
1644     /**
1645     * Returns a {@code BigDecimal} whose value is {@code (this /
1646     * divisor)}, and whose preferred scale is {@code (this.scale() -
1647     * divisor.scale())}; if the exact quotient cannot be
1648     * represented (because it has a non-terminating decimal
1649     * expansion) an {@code ArithmeticException} is thrown.
1650     *
1651     * @param divisor value by which this {@code BigDecimal} is to be divided.
1652     * @throws ArithmeticException if the exact quotient does not have a
1653     *         terminating decimal expansion

```

```

1654     * @return {@code this / divisor}
1655     * @since 1.5
1656     * @author Joseph D. Darcy
1657     */
1658     public BigDecimal divide(BigDecimal divisor) {
1659         /*
1660          * Handle zero cases first.
1661          */
1662         if (divisor.signum() == 0) {    // x/0
1663             if (this.signum() == 0)    // 0/0
1664                 throw new ArithmeticException("Division undefined"); // NaN
1665             throw new ArithmeticException("Division by zero");
1666         }
1667
1668         // Calculate preferred scale
1669         int preferredScale = saturateLong((long) this.scale - divisor.scale);
1670
1671         if (this.signum() == 0) // 0/y
1672             return zeroValueOf(preferredScale);
1673         else {
1674             /*
1675              * If the quotient this/divisor has a terminating decimal
1676              * expansion, the expansion can have no more than
1677              * (a.precision() + ceil(10*b.precision)/3) digits.
1678              * Therefore, create a MathContext object with this
1679              * precision and do a divide with the UNNECESSARY rounding
1680              * mode.
1681              */
1682             MathContext mc = new MathContext( (int)Math.min(this.precision() +
1683                                                         (long)Math.ceil(10.0*divisor.precision
1684                                                         ()/3.0),
1685                                                         Integer.MAX_VALUE),
1686                                                         RoundingMode.UNNECESSARY);
1687
1688             BigDecimal quotient;
1689             try {
1690                 quotient = this.divide(divisor, mc);
1691             } catch (ArithmeticException e) {
1692                 throw new ArithmeticException("Non-terminating decimal expansion; " +
1693                                                 "no exact representable decimal result.");
1694             }
1695
1696             int quotientScale = quotient.scale();
1697
1698             // divide(BigDecimal, mc) tries to adjust the quotient to
1699             // the desired one by removing trailing zeros; since the
1700             // exact divide method does not have an explicit digit
1701             // limit, we can add zeros too.
1702             if (preferredScale > quotientScale)
1703                 return quotient.setScale(preferredScale, ROUND_UNNECESSARY);
1704
1705             return quotient;
1706         }
1707     }

```



```

1706
1707 /**
1708  * Returns a {@code BigDecimal} whose value is {@code (this /
1709  * divisor)}, with rounding according to the context settings.
1710  *
1711  * @param divisor value by which this {@code BigDecimal} is to be divided.
1712  * @param mc the context to use.
1713  * @return {@code this / divisor}, rounded as necessary.
1714  * @throws ArithmeticException if the result is inexact but the
1715  *         rounding mode is {@code UNNECESSARY} or
1716  *         {@code mc.precision == 0} and the quotient has a
1717  *         non-terminating decimal expansion.
1718  * @since 1.5
1719  */
1720 public BigDecimal divide(BigDecimal divisor, MathContext mc) {
1721     int mcp = mc.precision;
1722     if (mcp == 0)
1723         return divide(divisor);
1724
1725     BigDecimal dividend = this;
1726     long preferredScale = (long)dividend.scale - divisor.scale;
1727     // Now calculate the answer. We use the existing
1728     // divide-and-round method, but as this rounds to scale we have
1729     // to normalize the values here to achieve the desired result.
1730     // For x/y we first handle y=0 and x=0, and then normalize x and
1731     // y to give x' and y' with the following constraints:
1732     // (a) 0.1 <= x' < 1
1733     // (b) x' <= y' < 10*x'
1734     // Dividing x'/y' with the required scale set to mc.precision then
1735     // will give a result in the range 0.1 to 1 rounded to exactly
1736     // the right number of digits (except in the case of a result of
1737     // 1.000... which can arise when x=y, or when rounding overflows
1738     // The 1.000... case will reduce properly to 1.
1739     if (divisor.signum() == 0) { // x/0
1740         if (dividend.signum() == 0) // 0/0
1741             throw new ArithmeticException("Division undefined"); // NaN
1742         throw new ArithmeticException("Division by zero");
1743     }
1744     if (dividend.signum() == 0) // 0/y
1745         return zeroValueOf(saturateLong(preferredScale));
1746     int xscale = dividend.precision();
1747     int yscale = divisor.precision();
1748     if (dividend.intCompact != INFLATED) {
1749         if (divisor.intCompact != INFLATED) {
1750             return divide(dividend.intCompact, xscale, divisor.intCompact, yscale,
1751                             preferredScale, mc);
1752         } else {
1753             return divide(dividend.intCompact, xscale, divisor.intVal, yscale, preferredScale,
1754                             mc);
1755         }
1756     } else {
1757         if (divisor.intCompact != INFLATED) {
1758             return divide(dividend.intVal, xscale, divisor.intCompact, yscale, preferredScale,
1759                             mc);
1760         } else {
1761             return divide(dividend.intVal, xscale, divisor.intVal, yscale, preferredScale,
1762                             mc);
1763         }
1764     }
1765 }

```

```

        mc);
1757     } else {
1758         return divide(dividend.intVal, xscale, divisor.intVal, yscale, preferredScale, mc);
1759     }
1760 }
1761 }
1762
1763 /**
1764  * Returns a {@code BigDecimal} whose value is the integer part
1765  * of the quotient {@code (this / divisor)} rounded down. The
1766  * preferred scale of the result is {@code (this.scale() -
1767  * divisor.scale())}.
1768  *
1769  * @param divisor value by which this {@code BigDecimal} is to be divided.
1770  * @return The integer part of {@code this / divisor}.
1771  * @throws ArithmeticException if {@code divisor==0}
1772  * @since 1.5
1773  */
1774 public BigDecimal divideToIntegralValue(BigDecimal divisor) {
1775     // Calculate preferred scale
1776     int preferredScale = saturateLong((long) this.scale - divisor.scale);
1777     if (this.compareMagnitude(divisor) < 0) {
1778         // much faster when this << divisor
1779         return zeroValueOf(preferredScale);
1780     }
1781
1782     if (this.signum() == 0 && divisor.signum() != 0)
1783         return this.setScale(preferredScale, ROUND_UNNECESSARY);
1784
1785     // Perform a divide with enough digits to round to a correct
1786     // integer value; then remove any fractional digits
1787
1788     int maxDigits = (int) Math.min(this.precision() +
1789                                   (long) Math.ceil(10.0 * divisor.precision() / 3.0) +
1790                                   Math.abs((long) this.scale() - divisor.scale()) + 2,
1791                                   Integer.MAX_VALUE);
1792     BigDecimal quotient = this.divide(divisor, new MathContext(maxDigits,
1793                                                                RoundingMode.DOWN));
1794     if (quotient.scale > 0) {
1795         quotient = quotient.setScale(0, RoundingMode.DOWN);
1796         quotient = stripZerosToMatchScale(quotient.intVal, quotient.intCompact, quotient.scale,
1797                                           preferredScale);
1798     }
1799     if (quotient.scale < preferredScale) {
1800         // pad with zeros if necessary
1801         quotient = quotient.setScale(preferredScale, ROUND_UNNECESSARY);
1802     }
1803
1804     return quotient;
1805 }
1806
1807 /**

```

```

1808     * Returns a {@code BigDecimal} whose value is the integer part
1809     * of {@code (this / divisor)}. Since the integer part of the
1810     * exact quotient does not depend on the rounding mode, the
1811     * rounding mode does not affect the values returned by this
1812     * method. The preferred scale of the result is
1813     * {@code (this.scale() - divisor.scale())}. An
1814     * {@code ArithmeticException} is thrown if the integer part of
1815     * the exact quotient needs more than {@code mc.precision}
1816     * digits.
1817     *
1818     * @param divisor value by which this {@code BigDecimal} is to be divided.
1819     * @param mc the context to use.
1820     * @return The integer part of {@code this / divisor}.
1821     * @throws ArithmeticException if {@code divisor==0}
1822     * @throws ArithmeticException if {@code mc.precision} {@code >} 0 and the result
1823     *         requires a precision of more than {@code mc.precision} digits.
1824     * @since 1.5
1825     * @author Joseph D. Darcy
1826     */
1827     public BigDecimal divideToIntegralValue(BigDecimal divisor, MathContext mc) {
1828         if (mc.precision == 0 || // exact result
1829             (this.compareMagnitude(divisor) < 0)) // zero result
1830             return divideToIntegralValue(divisor);
1831
1832         // Calculate preferred scale
1833         int preferredScale = saturateLong((long)this.scale - divisor.scale);
1834
1835         /*
1836          * Perform a normal divide to mc.precision digits. If the
1837          * remainder has absolute value less than the divisor, the
1838          * integer portion of the quotient fits into mc.precision
1839          * digits. Next, remove any fractional digits from the
1840          * quotient and adjust the scale to the preferred value.
1841          */
1842         BigDecimal result = this.divide(divisor, new MathContext(mc.precision, RoundingMode.DOWN));
1843
1844         if (result.scale() < 0) {
1845             /*
1846              * Result is an integer. See if quotient represents the
1847              * full integer portion of the exact quotient; if it does,
1848              * the computed remainder will be less than the divisor.
1849              */
1850             BigDecimal product = result.multiply(divisor);
1851             // If the quotient is the full integer value,
1852             // |dividend-product| < |divisor|.
1853             if (this.subtract(product).compareMagnitude(divisor) >= 0) {
1854                 throw new ArithmeticException("Division impossible");
1855             }
1856         } else if (result.scale() > 0) {
1857             /*
1858              * Integer portion of quotient will fit into precision
1859              * digits; recompute quotient to scale 0 to avoid double
1860              * rounding and then try to adjust, if necessary.

```

```

1861         */
1862         result = result.setScale(0, RoundingMode.DOWN);
1863     }
1864     // else result.scale() == 0;
1865
1866     int precisionDiff;
1867     if ((preferredScale > result.scale()) &&
1868         (precisionDiff = mc.precision - result.precision()) > 0) {
1869         return result.setScale(result.scale() +
1870                               Math.min(precisionDiff, preferredScale - result.scale) );
1871     } else {
1872         return stripZerosToMatchScale(result.intVal,result.intCompact,result.scale,
1873                                       preferredScale);
1874     }
1875 }
1876
1877 /**
1878  * Returns a {@code BigDecimal} whose value is {@code (this % divisor)}.
1879  *
1880  * <p>The remainder is given by
1881  * {@code this.subtract(this.divideToIntegralValue(divisor).multiply(divisor))}.
1882  * Note that this is not the modulo operation (the result can be
1883  * negative).
1884  *
1885  * @param divisor value by which this {@code BigDecimal} is to be divided.
1886  * @return {@code this % divisor}.
1887  * @throws ArithmeticException if {@code divisor==0}
1888  * @since 1.5
1889  */
1890 public BigDecimal remainder(BigDecimal divisor) {
1891     BigDecimal divrem[] = this.divideAndRemainder(divisor);
1892     return divrem[1];
1893 }
1894
1895 /**
1896  * Returns a {@code BigDecimal} whose value is {@code (this %
1897  * divisor)}, with rounding according to the context settings.
1898  * The {@code MathContext} settings affect the implicit divide
1899  * used to compute the remainder. The remainder computation
1900  * itself is by definition exact. Therefore, the remainder may
1901  * contain more than {@code mc.getPrecision()} digits.
1902  *
1903  * <p>The remainder is given by
1904  * {@code this.subtract(this.divideToIntegralValue(divisor,
1905  * mc).multiply(divisor))}. Note that this is not the modulo
1906  * operation (the result can be negative).
1907  *
1908  * @param divisor value by which this {@code BigDecimal} is to be divided.
1909  * @param mc the context to use.
1910  * @return {@code this % divisor}, rounded as necessary.
1911  * @throws ArithmeticException if {@code divisor==0}
1912  * @throws ArithmeticException if the result is inexact but the

```

```

1913 *      rounding mode is {@code UNNECESSARY}, or {@code mc.precision}
1914 *      {@code >} 0 and the result of {@code this.divideToIntegralValue(divisor)} would
1915 *      require a precision of more than {@code mc.precision} digits.
1916 * @see   #divideToIntegralValue(java.math.BigDecimal, java.math.MathContext)
1917 * @since 1.5
1918 */
1919 public BigDecimal remainder(BigDecimal divisor, MathContext mc) {
1920     BigDecimal divrem[] = this.divideAndRemainder(divisor, mc);
1921     return divrem[1];
1922 }
1923
1924 /**
1925  * Returns a two-element {@code BigDecimal} array containing the
1926  * result of {@code divideToIntegralValue} followed by the result of
1927  * {@code remainder} on the two operands.
1928  *
1929  * <p>Note that if both the integer quotient and remainder are
1930  * needed, this method is faster than using the
1931  * {@code divideToIntegralValue} and {@code remainder} methods
1932  * separately because the division need only be carried out once.
1933  *
1934  * @param divisor value by which this {@code BigDecimal} is to be divided,
1935  *        and the remainder computed.
1936  * @return a two element {@code BigDecimal} array: the quotient
1937  *         (the result of {@code divideToIntegralValue}) is the initial element
1938  *         and the remainder is the final element.
1939  * @throws ArithmeticException if {@code divisor==0}
1940  * @see   #divideToIntegralValue(java.math.BigDecimal, java.math.MathContext)
1941  * @see   #remainder(java.math.BigDecimal, java.math.MathContext)
1942  * @since 1.5
1943  */
1944 public BigDecimal[] divideAndRemainder(BigDecimal divisor) {
1945     // we use the identity  $x = i * y + r$  to determine r
1946     BigDecimal[] result = new BigDecimal[2];
1947
1948     result[0] = this.divideToIntegralValue(divisor);
1949     result[1] = this.subtract(result[0].multiply(divisor));
1950     return result;
1951 }
1952
1953 /**
1954  * Returns a two-element {@code BigDecimal} array containing the
1955  * result of {@code divideToIntegralValue} followed by the result of
1956  * {@code remainder} on the two operands calculated with rounding
1957  * according to the context settings.
1958  *
1959  * <p>Note that if both the integer quotient and remainder are
1960  * needed, this method is faster than using the
1961  * {@code divideToIntegralValue} and {@code remainder} methods
1962  * separately because the division need only be carried out once.
1963  *
1964  * @param divisor value by which this {@code BigDecimal} is to be divided,
1965  *        and the remainder computed.

```

```

1966     * @param mc the context to use.
1967     * @return a two element {@code BigDecimal} array: the quotient
1968     *         (the result of {@code divideToIntegralValue}) is the
1969     *         initial element and the remainder is the final element.
1970     * @throws ArithmeticException if {@code divisor==0}
1971     * @throws ArithmeticException if the result is inexact but the
1972     *         rounding mode is {@code UNNECESSARY}, or {@code mc.precision}
1973     *         {@literal >} 0 and the result of {@code this.divideToIntegralValue(divisor)} would
1974     *         require a precision of more than {@code mc.precision} digits.
1975     * @see    #divideToIntegralValue(java.math.BigDecimal, java.math.MathContext)
1976     * @see    #remainder(java.math.BigDecimal, java.math.MathContext)
1977     * @since 1.5
1978     */
1979     public BigDecimal[] divideAndRemainder(BigDecimal divisor, MathContext mc) {
1980         if (mc.precision == 0)
1981             return divideAndRemainder(divisor);
1982
1983         BigDecimal[] result = new BigDecimal[2];
1984         BigDecimal lhs = this;
1985
1986         result[0] = lhs.divideToIntegralValue(divisor, mc);
1987         result[1] = lhs.subtract(result[0].multiply(divisor));
1988         return result;
1989     }
1990
1991     /**
1992     * Returns a {@code BigDecimal} whose value is
1993     *  $\langle \text{this} \rangle^n$ , The power is computed exactly, to
1994     * unlimited precision.
1995     *
1996     * 

The parameter {@code n} must be in the range 0 through
1997     * 999999999, inclusive. {@code ZERO.pow(0)} returns {@link
1998     * #ONE}.


1999     *
2000     * Note that future releases may expand the allowable exponent
2001     * range of this method.
2002     *
2003     * @param n power to raise this {@code BigDecimal} to.
2004     * @return  $\langle \text{this} \rangle^n$ 
2005     * @throws ArithmeticException if {@code n} is out of range.
2006     * @since 1.5
2007     */
2008     public BigDecimal pow(int n) {
2009         if (n < 0 || n > 999999999)
2010             throw new ArithmeticException("Invalid operation");
2011         // No need to calculate pow(n) if result will over/underflow.
2012         // Don't attempt to support "supernormal" numbers.
2013         int newScale = checkScale((long)scale * n);
2014         return new BigDecimal(this.inflated().pow(n), newScale);
2015     }
2016
2017
2018     /**

```

```

2019 * Returns a {@code BigDecimal} whose value is
2020 * <tt>(this<sup>n</sup></tt>. The current implementation uses
2021 * the core algorithm defined in ANSI standard X3.274-1996 with
2022 * rounding according to the context settings. In general, the
2023 * returned numerical value is within two ulps of the exact
2024 * numerical value for the chosen precision. Note that future
2025 * releases may use a different algorithm with a decreased
2026 * allowable error bound and increased allowable exponent range.
2027 *
2028 * <p>The X3.274-1996 algorithm is:
2029 *
2030 * <ul>
2031 * <li> An {@code ArithmeticException} exception is thrown if
2032 * <ul>
2033 * <li>{@code abs(n) > 999999999}
2034 * <li>{@code mc.precision == 0} and {@code n < 0}
2035 * <li>{@code mc.precision > 0} and {@code n} has more than
2036 * {@code mc.precision} decimal digits
2037 * </ul>
2038 *
2039 * <li> if {@code n} is zero, {@link #ONE} is returned even if
2040 * {@code this} is zero, otherwise
2041 * <ul>
2042 * <li> if {@code n} is positive, the result is calculated via
2043 * the repeated squaring technique into a single accumulator.
2044 * The individual multiplications with the accumulator use the
2045 * same math context settings as in {@code mc} except for a
2046 * precision increased to {@code mc.precision + elength + 1}
2047 * where {@code elength} is the number of decimal digits in
2048 * {@code n}.
2049 *
2050 * <li> if {@code n} is negative, the result is calculated as if
2051 * {@code n} were positive; this value is then divided into one
2052 * using the working precision specified above.
2053 *
2054 * <li> The final value from either the positive or negative case
2055 * is then rounded to the destination precision.
2056 * </ul>
2057 * </ul>
2058 *
2059 * @param n power to raise this {@code BigDecimal} to.
2060 * @param mc the context to use.
2061 * @return <tt>this<sup>n</sup></tt> using the ANSI standard X3.274-1996
2062 * algorithm
2063 * @throws ArithmeticException if the result is inexact but the
2064 * rounding mode is {@code UNNECESSARY}, or {@code n} is out
2065 * of range.
2066 * @since 1.5
2067 */
2068 public BigDecimal pow(int n, MathContext mc) {
2069     if (mc.precision == 0)
2070         return pow(n);
2071     if (n < -999999999 || n > 999999999)

```



```

2072         throw new ArithmeticException("Invalid operation");
2073     if (n == 0)
2074         return ONE; // x**0 == 1 in X3.274
2075     BigDecimal lhs = this;
2076     MathContext workmc = mc; // working settings
2077     int mag = Math.abs(n); // magnitude of n
2078     if (mc.precision > 0) {
2079         int elength = longDigitLength(mag); // length of n in digits
2080         if (elength > mc.precision) // X3.274 rule
2081             throw new ArithmeticException("Invalid operation");
2082         workmc = new MathContext(mc.precision + elength + 1,
2083                                 mc.roundingMode);
2084     }
2085     // ready to carry out power calculation...
2086     BigDecimal acc = ONE; // accumulator
2087     boolean seenbit = false; // set once we've seen a 1-bit
2088     for (int i=1;;i++) { // for each bit [top bit ignored]
2089         mag += mag; // shift left 1 bit
2090         if (mag < 0) { // top bit is set
2091             seenbit = true; // OK, we're off
2092             acc = acc.multiply(lhs, workmc); // acc=acc*x
2093         }
2094         if (i == 31)
2095             break; // that was the last bit
2096         if (seenbit)
2097             acc=acc.multiply(acc, workmc); // acc=acc*acc [square]
2098             // else (!seenbit) no point in squaring ONE
2099     }
2100     // if negative n, calculate the reciprocal using working precision
2101     if (n < 0) // [hence mc.precision>0]
2102         acc=ONE.divide(acc, workmc);
2103     // round to final precision and strip zeros
2104     return doRound(acc, mc);
2105 }
2106
2107 /**
2108  * Returns a {@code BigDecimal} whose value is the absolute value
2109  * of this {@code BigDecimal}, and whose scale is
2110  * {@code this.scale()}.
2111  *
2112  * @return {@code abs(this)}
2113  */
2114 public BigDecimal abs() {
2115     return (signum() < 0 ? negate() : this);
2116 }
2117
2118 /**
2119  * Returns a {@code BigDecimal} whose value is the absolute value
2120  * of this {@code BigDecimal}, with rounding according to the
2121  * context settings.
2122  *
2123  * @param mc the context to use.
2124  * @return {@code abs(this)}, rounded as necessary.

```



```

2125     * @throws ArithmeticException if the result is inexact but the
2126     *         rounding mode is {@code UNNECESSARY}.
2127     * @since 1.5
2128     */
2129     public BigDecimal abs(MathContext mc) {
2130         return (signum() < 0 ? negate(mc) : plus(mc));
2131     }
2132
2133     /**
2134     * Returns a {@code BigDecimal} whose value is {@code (-this)},
2135     * and whose scale is {@code this.scale()}.
2136     *
2137     * @return {@code -this}.
2138     */
2139     public BigDecimal negate() {
2140         if (intCompact == INFLATED) {
2141             return new BigDecimal(intVal.negate(), INFLATED, scale, precision);
2142         } else {
2143             return valueOf(-intCompact, scale, precision);
2144         }
2145     }
2146
2147     /**
2148     * Returns a {@code BigDecimal} whose value is {@code (-this)},
2149     * with rounding according to the context settings.
2150     *
2151     * @param mc the context to use.
2152     * @return {@code -this}, rounded as necessary.
2153     * @throws ArithmeticException if the result is inexact but the
2154     *         rounding mode is {@code UNNECESSARY}.
2155     * @since 1.5
2156     */
2157     public BigDecimal negate(MathContext mc) {
2158         return negate().plus(mc);
2159     }
2160
2161     /**
2162     * Returns a {@code BigDecimal} whose value is {@code (+this)}, and whose
2163     * scale is {@code this.scale()}.
2164     *
2165     * <p>This method, which simply returns this {@code BigDecimal}
2166     * is included for symmetry with the unary minus method {@link
2167     * #negate()}.
2168     *
2169     * @return {@code this}.
2170     * @see #negate()
2171     * @since 1.5
2172     */
2173     public BigDecimal plus() {
2174         return this;
2175     }
2176
2177     /**

```

```

2178     * Returns a {@code BigDecimal} whose value is {@code (+this)},
2179     * with rounding according to the context settings.
2180     *
2181     * <p>The effect of this method is identical to that of the {@link
2182     * #round(MathContext)} method.
2183     *
2184     * @param mc the context to use.
2185     * @return {@code this}, rounded as necessary. A zero result will
2186     *         have a scale of 0.
2187     * @throws ArithmeticException if the result is inexact but the
2188     *         rounding mode is {@code UNNECESSARY}.
2189     * @see    #round(MathContext)
2190     * @since 1.5
2191     */
2192     public BigDecimal plus(MathContext mc) {
2193         if (mc.precision == 0)                // no rounding please
2194             return this;
2195         return doRound(this, mc);
2196     }
2197
2198     /**
2199     * Returns the signum function of this {@code BigDecimal}.
2200     *
2201     * @return -1, 0, or 1 as the value of this {@code BigDecimal}
2202     *         is negative, zero, or positive.
2203     */
2204     public int signum() {
2205         return (intCompact != INFLATED)?
2206             Long.signum(intCompact):
2207             intVal.signum();
2208     }
2209
2210     /**
2211     * Returns the <i>scale</i> of this {@code BigDecimal}. If zero
2212     * or positive, the scale is the number of digits to the right of
2213     * the decimal point. If negative, the unscaled value of the
2214     * number is multiplied by ten to the power of the negation of the
2215     * scale. For example, a scale of {@code -3} means the unscaled
2216     * value is multiplied by 1000.
2217     *
2218     * @return the scale of this {@code BigDecimal}.
2219     */
2220     public int scale() {
2221         return scale;
2222     }
2223
2224     /**
2225     * Returns the <i>precision</i> of this {@code BigDecimal}. (The
2226     * precision is the number of digits in the unscaled value.)
2227     *
2228     * <p>The precision of a zero value is 1.
2229     *
2230     * @return the precision of this {@code BigDecimal}.

```

```

2231     * @since 1.5
2232     */
2233     public int precision() {
2234         int result = precision;
2235         if (result == 0) {
2236             long s = intCompact;
2237             if (s != INFLATED)
2238                 result = longDigitLength(s);
2239             else
2240                 result = bigDigitLength(intVal);
2241             precision = result;
2242         }
2243         return result;
2244     }
2245
2246
2247     /**
2248     * Returns a {@code BigInteger} whose value is the <i>unscaled
2249     * value</i> of this {@code BigDecimal}. (Computes <tt>(this *
2250     * 10<sup>this.scale()</sup></tt>.)
2251     *
2252     * @return the unscaled value of this {@code BigDecimal}.
2253     * @since 1.2
2254     */
2255     public BigInteger unscaledValue() {
2256         return this.inflated();
2257     }
2258
2259     // Rounding Modes
2260
2261     /**
2262     * Rounding mode to round away from zero. Always increments the
2263     * digit prior to a nonzero discarded fraction. Note that this rounding
2264     * mode never decreases the magnitude of the calculated value.
2265     */
2266     public final static int ROUND_UP = 0;
2267
2268     /**
2269     * Rounding mode to round towards zero. Never increments the digit
2270     * prior to a discarded fraction (i.e., truncates). Note that this
2271     * rounding mode never increases the magnitude of the calculated value.
2272     */
2273     public final static int ROUND_DOWN = 1;
2274
2275     /**
2276     * Rounding mode to round towards positive infinity. If the
2277     * {@code BigDecimal} is positive, behaves as for
2278     * {@code ROUND_UP}; if negative, behaves as for
2279     * {@code ROUND_DOWN}. Note that this rounding mode never
2280     * decreases the calculated value.
2281     */
2282     public final static int ROUND_CEILING = 2;
2283

```

```

2284 /**
2285  * Rounding mode to round towards negative infinity. If the
2286  * {@code BigDecimal} is positive, behave as for
2287  * {@code ROUND_DOWN}; if negative, behave as for
2288  * {@code ROUND_UP}. Note that this rounding mode never
2289  * increases the calculated value.
2290  */
2291 public final static int ROUND_FLOOR = 3;
2292
2293 /**
2294  * Rounding mode to round towards {@literal "nearest neighbor"}
2295  * unless both neighbors are equidistant, in which case round up.
2296  * Behaves as for {@code ROUND_UP} if the discarded fraction is
2297  *  $\geq 0.5$ ; otherwise, behaves as for {@code ROUND_DOWN}. Note
2298  * that this is the rounding mode that most of us were taught in
2299  * grade school.
2300  */
2301 public final static int ROUND_HALF_UP = 4;
2302
2303 /**
2304  * Rounding mode to round towards {@literal "nearest neighbor"}
2305  * unless both neighbors are equidistant, in which case round
2306  * down. Behaves as for {@code ROUND_UP} if the discarded
2307  * fraction is  $> 0.5$ ; otherwise, behaves as for
2308  * {@code ROUND_DOWN}.
2309  */
2310 public final static int ROUND_HALF_DOWN = 5;
2311
2312 /**
2313  * Rounding mode to round towards the {@literal "nearest neighbor"}
2314  * unless both neighbors are equidistant, in which case, round
2315  * towards the even neighbor. Behaves as for
2316  * {@code ROUND_HALF_UP} if the digit to the left of the
2317  * discarded fraction is odd; behaves as for
2318  * {@code ROUND_HALF_DOWN} if it's even. Note that this is the
2319  * rounding mode that minimizes cumulative error when applied
2320  * repeatedly over a sequence of calculations.
2321  */
2322 public final static int ROUND_HALF_EVEN = 6;
2323
2324 /**
2325  * Rounding mode to assert that the requested operation has an exact
2326  * result, hence no rounding is necessary. If this rounding mode is
2327  * specified on an operation that yields an inexact result, an
2328  * {@code ArithmeticException} is thrown.
2329  */
2330 public final static int ROUND_UNNECESSARY = 7;
2331
2332
2333 // Scaling/Rounding Operations
2334
2335 /**
2336  * Returns a {@code BigDecimal} rounded according to the

```

```

2337 * {@code MathContext} settings. If the precision setting is 0 then
2338 * no rounding takes place.
2339 *
2340 * <p>The effect of this method is identical to that of the
2341 * {@link #plus(MathContext)} method.
2342 *
2343 * @param mc the context to use.
2344 * @return a {@code BigDecimal} rounded according to the
2345 *         {@code MathContext} settings.
2346 * @throws ArithmeticException if the rounding mode is
2347 *         {@code UNNECESSARY} and the
2348 *         {@code BigDecimal} operation would require rounding.
2349 * @see    #plus(MathContext)
2350 * @since 1.5
2351 */
2352 public BigDecimal round(MathContext mc) {
2353     return plus(mc);
2354 }
2355
2356 /**
2357  * Returns a {@code BigDecimal} whose scale is the specified
2358  * value, and whose unscaled value is determined by multiplying or
2359  * dividing this {@code BigDecimal}'s unscaled value by the
2360  * appropriate power of ten to maintain its overall value. If the
2361  * scale is reduced by the operation, the unscaled value must be
2362  * divided (rather than multiplied), and the value may be changed;
2363  * in this case, the specified rounding mode is applied to the
2364  * division.
2365  *
2366  * <p>Note that since BigDecimal objects are immutable, calls of
2367  * this method do <i>not</i> result in the original object being
2368  * modified, contrary to the usual convention of having methods
2369  * named <tt>set<i>X</i></tt> mutate field <i>{@code X}</i>.
2370  * Instead, {@code setScale} returns an object with the proper
2371  * scale; the returned object may or may not be newly allocated.
2372  *
2373  * @param newScale scale of the {@code BigDecimal} value to be returned.
2374  * @param roundingMode The rounding mode to apply.
2375  * @return a {@code BigDecimal} whose scale is the specified value,
2376  *         and whose unscaled value is determined by multiplying or
2377  *         dividing this {@code BigDecimal}'s unscaled value by the
2378  *         appropriate power of ten to maintain its overall value.
2379  * @throws ArithmeticException if {@code roundingMode==UNNECESSARY}
2380  *         and the specified scaling operation would require
2381  *         rounding.
2382  * @see    RoundingMode
2383  * @since 1.5
2384  */
2385 public BigDecimal setScale(int newScale, RoundingMode roundingMode) {
2386     return setScale(newScale, roundingMode.oldMode);
2387 }
2388
2389 /**

```

```

2390 * Returns a {@code BigDecimal} whose scale is the specified
2391 * value, and whose unscaled value is determined by multiplying or
2392 * dividing this {@code BigDecimal}'s unscaled value by the
2393 * appropriate power of ten to maintain its overall value. If the
2394 * scale is reduced by the operation, the unscaled value must be
2395 * divided (rather than multiplied), and the value may be changed;
2396 * in this case, the specified rounding mode is applied to the
2397 * division.
2398 *
2399 * <p>Note that since BigDecimal objects are immutable, calls of
2400 * this method do <i>not</i> result in the original object being
2401 * modified, contrary to the usual convention of having methods
2402 * named <tt>set<i>X</i></tt> mutate field <i>{@code X}</i>.
2403 * Instead, {@code setScale} returns an object with the proper
2404 * scale; the returned object may or may not be newly allocated.
2405 *
2406 * <p>The new {@link #setScale(int, RoundingMode)} method should
2407 * be used in preference to this legacy method.
2408 *
2409 * @param newScale scale of the {@code BigDecimal} value to be returned.
2410 * @param roundingMode The rounding mode to apply.
2411 * @return a {@code BigDecimal} whose scale is the specified value,
2412 *         and whose unscaled value is determined by multiplying or
2413 *         dividing this {@code BigDecimal}'s unscaled value by the
2414 *         appropriate power of ten to maintain its overall value.
2415 * @throws ArithmeticException if {@code roundingMode==ROUND_UNNECESSARY}
2416 *         and the specified scaling operation would require
2417 *         rounding.
2418 * @throws IllegalArgumentException if {@code roundingMode} does not
2419 *         represent a valid rounding mode.
2420 * @see    #ROUND_UP
2421 * @see    #ROUND_DOWN
2422 * @see    #ROUND_CEILING
2423 * @see    #ROUND_FLOOR
2424 * @see    #ROUND_HALF_UP
2425 * @see    #ROUND_HALF_DOWN
2426 * @see    #ROUND_HALF_EVEN
2427 * @see    #ROUND_UNNECESSARY
2428 */
2429 public BigDecimal setScale(int newScale, int roundingMode) {
2430     if (roundingMode < ROUND_UP || roundingMode > ROUND_UNNECESSARY)
2431         throw new IllegalArgumentException("Invalid rounding mode");
2432
2433     int oldScale = this.scale;
2434     if (newScale == oldScale) // easy case
2435         return this;
2436     if (this.signum() == 0) // zero can have any scale
2437         return zeroValueOf(newScale);
2438     if (this.intCompact != INFLATED) {
2439         long rs = this.intCompact;
2440         if (newScale > oldScale) {
2441             int raise = checkScale((long) newScale - oldScale);
2442             if ((rs = longMultiplyPowerTen(rs, raise)) != INFLATED) {

```

```

2443         return valueOf(rs,newScale);
2444     }
2445     BigInteger rb = bigMultiplyPowerTen(raise);
2446     return new BigDecimal(rb, INFLATED, newScale, (precision > 0) ? precision + raise :
        0);
2447 } else {
2448     // newScale < oldScale -- drop some digits
2449     // Can't predict the precision due to the effect of rounding.
2450     int drop = checkScale((long) oldScale - newScale);
2451     if (drop < LONG_TEN_POWERS_TABLE.length) {
2452         return divideAndRound(rs, LONG_TEN_POWERS_TABLE[drop], newScale, roundingMode,
            newScale);
2453     } else {
2454         return divideAndRound(this.inflated(), bigTenToThe(drop), newScale,
            roundingMode, newScale);
2455     }
2456 }
2457 } else {
2458     if (newScale > oldScale) {
2459         int raise = checkScale((long) newScale - oldScale);
2460         BigInteger rb = bigMultiplyPowerTen(this.intVal,raise);
2461         return new BigDecimal(rb, INFLATED, newScale, (precision > 0) ? precision + raise :
            0);
2462     } else {
2463         // newScale < oldScale -- drop some digits
2464         // Can't predict the precision due to the effect of rounding.
2465         int drop = checkScale((long) oldScale - newScale);
2466         if (drop < LONG_TEN_POWERS_TABLE.length)
2467             return divideAndRound(this.intVal, LONG_TEN_POWERS_TABLE[drop], newScale,
                roundingMode,
2468                 newScale);
2469         else
2470             return divideAndRound(this.intVal, bigTenToThe(drop), newScale, roundingMode,
                newScale);
2471     }
2472 }
2473 }
2474
2475 /**
2476  * Returns a {@code BigDecimal} whose scale is the specified
2477  * value, and whose value is numerically equal to this
2478  * {@code BigDecimal}'s. Throws an {@code ArithmeticException}
2479  * if this is not possible.
2480  *
2481  * <p>This call is typically used to increase the scale, in which
2482  * case it is guaranteed that there exists a {@code BigDecimal}
2483  * of the specified scale and the correct value. The call can
2484  * also be used to reduce the scale if the caller knows that the
2485  * {@code BigDecimal} has sufficiently many zeros at the end of
2486  * its fractional part (i.e., factors of ten in its integer value)
2487  * to allow for the rescaling without changing its value.
2488  *
2489  * <p>This method returns the same result as the two-argument

```

```

2490     * versions of {@code setScale}, but saves the caller the trouble
2491     * of specifying a rounding mode in cases where it is irrelevant.
2492     *
2493     * <p>Note that since {@code BigDecimal} objects are immutable,
2494     * calls of this method do <i>not</i> result in the original
2495     * object being modified, contrary to the usual convention of
2496     * having methods named <tt>set<i>X</i></tt> mutate field
2497     * <i>{@code X}</i>. Instead, {@code setScale} returns an
2498     * object with the proper scale; the returned object may or may
2499     * not be newly allocated.
2500     *
2501     * @param newScale scale of the {@code BigDecimal} value to be returned.
2502     * @return a {@code BigDecimal} whose scale is the specified value, and
2503     *         whose unscaled value is determined by multiplying or dividing
2504     *         this {@code BigDecimal}'s unscaled value by the appropriate
2505     *         power of ten to maintain its overall value.
2506     * @throws ArithmeticException if the specified scaling operation would
2507     *         require rounding.
2508     * @see #setScale(int, int)
2509     * @see #setScale(int, RoundingMode)
2510     */
2511     public BigDecimal setScale(int newScale) {
2512         return setScale(newScale, ROUND_UNNECESSARY);
2513     }
2514
2515     // Decimal Point Motion Operations
2516
2517     /**
2518     * Returns a {@code BigDecimal} which is equivalent to this one
2519     * with the decimal point moved {@code n} places to the left. If
2520     * {@code n} is non-negative, the call merely adds {@code n} to
2521     * the scale. If {@code n} is negative, the call is equivalent
2522     * to {@code movePointRight(-n)}. The {@code BigDecimal}
2523     * returned by this call has value <tt>(this &times;
2524     * 10<sup>-n</sup></tt> and scale {@code max(this.scale()+n,
2525     * 0)}.
2526     *
2527     * @param n number of places to move the decimal point to the left.
2528     * @return a {@code BigDecimal} which is equivalent to this one with the
2529     *         decimal point moved {@code n} places to the left.
2530     * @throws ArithmeticException if scale overflows.
2531     */
2532     public BigDecimal movePointLeft(int n) {
2533         // Cannot use movePointRight(-n) in case of n==Integer.MIN_VALUE
2534         int newScale = checkScale((long)scale + n);
2535         BigDecimal num = new BigDecimal(intVal, intCompact, newScale, 0);
2536         return num.scale < 0 ? num.setScale(0, ROUND_UNNECESSARY) : num;
2537     }
2538
2539     /**
2540     * Returns a {@code BigDecimal} which is equivalent to this one
2541     * with the decimal point moved {@code n} places to the right.
2542     * If {@code n} is non-negative, the call merely subtracts

```



```

2543 * {@code n} from the scale. If {@code n} is negative, the call
2544 * is equivalent to {@code movePointLeft(-n)}. The
2545 * {@code BigDecimal} returned by this call has value <tt>(this
2546 * <math>\times 10^{<sup>n</sup></math></tt> and scale {@code max(this.scale()-n,
2547 * 0)}.
2548 *
2549 * @param n number of places to move the decimal point to the right.
2550 * @return a {@code BigDecimal} which is equivalent to this one
2551 *         with the decimal point moved {@code n} places to the right.
2552 * @throws ArithmeticException if scale overflows.
2553 */
2554 public BigDecimal movePointRight(int n) {
2555     // Cannot use movePointLeft(-n) in case of n==Integer.MIN_VALUE
2556     int newScale = checkScale((long)scale - n);
2557     BigDecimal num = new BigDecimal(intVal, intCompact, newScale, 0);
2558     return num.scale < 0 ? num.setScale(0, ROUND_UNNECESSARY) : num;
2559 }
2560
2561 /**
2562 * Returns a BigDecimal whose numerical value is equal to
2563 * ({@code this} *  $10^{<sup>n</sup>}$ ). The scale of
2564 * the result is {@code (this.scale() - n)}.
2565 *
2566 * @param n the exponent power of ten to scale by
2567 * @return a BigDecimal whose numerical value is equal to
2568 *         ({@code this} *  $10^{<sup>n</sup>}$ )
2569 * @throws ArithmeticException if the scale would be
2570 *         outside the range of a 32-bit integer.
2571 *
2572 * @since 1.5
2573 */
2574 public BigDecimal scaleByPowerOfTen(int n) {
2575     return new BigDecimal(intVal, intCompact,
2576         checkScale((long)scale - n), precision);
2577 }
2578
2579 /**
2580 * Returns a {@code BigDecimal} which is numerically equal to
2581 * this one but with any trailing zeros removed from the
2582 * representation. For example, stripping the trailing zeros from
2583 * the {@code BigDecimal} value {@code 600.0}, which has
2584 * [{@code BigInteger}, {@code scale}] components equals to
2585 * [6000, 1], yields {@code 6E2} with [{@code BigInteger},
2586 * {@code scale}] components equals to [6, -2]. If
2587 * this BigDecimal is numerically equal to zero, then
2588 * {@code BigDecimal.ZERO} is returned.
2589 *
2590 * @return a numerically equal {@code BigDecimal} with any
2591 *         trailing zeros removed.
2592 * @since 1.5
2593 */
2594 public BigDecimal stripTrailingZeros() {
2595     if (intCompact == 0 || (intVal != null && intVal.signum() == 0)) {

```

```

2596         return BigDecimal.ZERO;
2597     } else if (intCompact != INFLATED) {
2598         return createAndStripZerosToMatchScale(intCompact, scale, Long.MIN_VALUE);
2599     } else {
2600         return createAndStripZerosToMatchScale(intVal, scale, Long.MIN_VALUE);
2601     }
2602 }
2603
2604 // Comparison Operations
2605
2606 /**
2607  * Compares this {@code BigDecimal} with the specified
2608  * {@code BigDecimal}. Two {@code BigDecimal} objects that are
2609  * equal in value but have a different scale (like 2.0 and 2.00)
2610  * are considered equal by this method. This method is provided
2611  * in preference to individual methods for each of the six boolean
2612  * comparison operators ({@literal <}, {@literal ==},
2613  * {@literal >}, {@literal >=}, {@literal !=}, {@literal <=}). The
2614  * suggested idiom for performing these comparisons is:
2615  * {@code (x.compareTo(y)) << i>op</i>> (y)}, where
2616  * << i>op</i>> is one of the six comparison operators.
2617  *
2618  * @param val {@code BigDecimal} to which this {@code BigDecimal} is
2619  *             to be compared.
2620  * @return -1, 0, or 1 as this {@code BigDecimal} is numerically
2621  *         less than, equal to, or greater than {@code val}.
2622  */
2623 public int compareTo(BigDecimal val) {
2624     // Quick path for equal scale and non-inflated case.
2625     if (scale == val.scale) {
2626         long xs = intCompact;
2627         long ys = val.intCompact;
2628         if (xs != INFLATED && ys != INFLATED)
2629             return xs != ys ? ((xs > ys) ? 1 : -1) : 0;
2630     }
2631     int xsign = this.signum();
2632     int ysign = val.signum();
2633     if (xsign != ysign)
2634         return (xsign > ysign) ? 1 : -1;
2635     if (xsign == 0)
2636         return 0;
2637     int cmp = compareMagnitude(val);
2638     return (xsign > 0) ? cmp : -cmp;
2639 }
2640
2641 /**
2642  * Version of compareTo that ignores sign.
2643  */
2644 private int compareMagnitude(BigDecimal val) {
2645     // Match scales, avoid unnecessary inflation
2646     long ys = val.intCompact;
2647     long xs = this.intCompact;
2648     if (xs == 0)

```

```

2649         return (ys == 0) ? 0 : -1;
2650     if (ys == 0)
2651         return 1;
2652
2653     long sdiff = (long)this.scale - val.scale;
2654     if (sdiff != 0) {
2655         // Avoid matching scales if the (adjusted) exponents differ
2656         long xae = (long)this.precision() - this.scale; // [-1]
2657         long yae = (long)val.precision() - val.scale;   // [-1]
2658         if (xae < yae)
2659             return -1;
2660         if (xae > yae)
2661             return 1;
2662         BigInteger rb = null;
2663         if (sdiff < 0) {
2664             // The cases sdiff <= Integer.MIN_VALUE intentionally fall through.
2665             if (sdiff > Integer.MIN_VALUE &&
2666                 (xs == INFLATED ||
2667                  (xs = longMultiplyPowerTen(xs, (int)-sdiff)) == INFLATED) &&
2668                  ys == INFLATED) {
2669                 rb = bigMultiplyPowerTen((int)-sdiff);
2670                 return rb.compareMagnitude(val.intVal);
2671             }
2672         } else { // sdiff > 0
2673             // The cases sdiff > Integer.MAX_VALUE intentionally fall through.
2674             if (sdiff <= Integer.MAX_VALUE &&
2675                 (ys == INFLATED ||
2676                  (ys = longMultiplyPowerTen(ys, (int)sdiff)) == INFLATED) &&
2677                  xs == INFLATED) {
2678                 rb = val.bigMultiplyPowerTen((int)sdiff);
2679                 return this.intVal.compareMagnitude(rb);
2680             }
2681         }
2682     }
2683     if (xs != INFLATED)
2684         return (ys != INFLATED) ? longCompareMagnitude(xs, ys) : -1;
2685     else if (ys != INFLATED)
2686         return 1;
2687     else
2688         return this.intVal.compareMagnitude(val.intVal);
2689 }
2690
2691 /**
2692  * Compares this {@code BigDecimal} with the specified
2693  * {@code Object} for equality. Unlike {@link
2694  * #compareTo(BigDecimal) compareTo}, this method considers two
2695  * {@code BigDecimal} objects equal only if they are equal in
2696  * value and scale (thus 2.0 is not equal to 2.00 when compared by
2697  * this method).
2698  *
2699  * @param x {@code Object} to which this {@code BigDecimal} is
2700  *          to be compared.
2701  * @return {@code true} if and only if the specified {@code Object} is a

```

```

2702     *      {@code BigDecimal} whose value and scale are equal to this
2703     *      {@code BigDecimal}'s.
2704     * @see   #compareTo(java.math.BigDecimal)
2705     * @see   #hashCode
2706     */
2707     @Override
2708     public boolean equals(Object x) {
2709         if (!(x instanceof BigDecimal))
2710             return false;
2711         BigDecimal xDec = (BigDecimal) x;
2712         if (x == this)
2713             return true;
2714         if (scale != xDec.scale)
2715             return false;
2716         long s = this.intCompact;
2717         long xs = xDec.intCompact;
2718         if (s != INFLATED) {
2719             if (xs == INFLATED)
2720                 xs = compactValFor(xDec.intVal);
2721             return xs == s;
2722         } else if (xs != INFLATED)
2723             return xs == compactValFor(this.intVal);
2724
2725         return this.inflated().equals(xDec.inflated());
2726     }
2727
2728     /**
2729     * Returns the minimum of this {@code BigDecimal} and
2730     * {@code val}.
2731     *
2732     * @param val value with which the minimum is to be computed.
2733     * @return the {@code BigDecimal} whose value is the lesser of this
2734     *         {@code BigDecimal} and {@code val}. If they are equal,
2735     *         as defined by the {@link #compareTo(BigDecimal) compareTo}
2736     *         method, {@code this} is returned.
2737     * @see   #compareTo(java.math.BigDecimal)
2738     */
2739     public BigDecimal min(BigDecimal val) {
2740         return (compareTo(val) <= 0 ? this : val);
2741     }
2742
2743     /**
2744     * Returns the maximum of this {@code BigDecimal} and {@code val}.
2745     *
2746     * @param val value with which the maximum is to be computed.
2747     * @return the {@code BigDecimal} whose value is the greater of this
2748     *         {@code BigDecimal} and {@code val}. If they are equal,
2749     *         as defined by the {@link #compareTo(BigDecimal) compareTo}
2750     *         method, {@code this} is returned.
2751     * @see   #compareTo(java.math.BigDecimal)
2752     */
2753     public BigDecimal max(BigDecimal val) {
2754         return (compareTo(val) >= 0 ? this : val);

```

```

2755     }
2756
2757     // Hash Function
2758
2759     /**
2760      * Returns the hash code for this {@code BigDecimal}. Note that
2761      * two {@code BigDecimal} objects that are numerically equal but
2762      * differ in scale (like 2.0 and 2.00) will generally <i>not</i>
2763      * have the same hash code.
2764      *
2765      * @return hash code for this {@code BigDecimal}.
2766      * @see #equals(Object)
2767      */
2768     @Override
2769     public int hashCode() {
2770         if (intCompact != INFLATED) {
2771             long val2 = (intCompact < 0)? -intCompact : intCompact;
2772             int temp = (int)( ((int)(val2 >>> 32)) * 31 +
2773                             (val2 & LONG_MASK));
2774             return 31*((intCompact < 0) ?-temp:temp) + scale;
2775         } else
2776             return 31*intVal.hashCode() + scale;
2777     }
2778
2779     // Format Converters
2780
2781     /**
2782      * Returns the string representation of this {@code BigDecimal},
2783      * using scientific notation if an exponent is needed.
2784      *
2785      * <p>A standard canonical string form of the {@code BigDecimal}
2786      * is created as though by the following steps: first, the
2787      * absolute value of the unscaled value of the {@code BigDecimal}
2788      * is converted to a string in base ten using the characters
2789      * {@code '0'} through {@code '9'} with no leading zeros (except
2790      * if its value is zero, in which case a single {@code '0'}
2791      * character is used).
2792      *
2793      * <p>Next, an <i>adjusted exponent</i> is calculated; this is the
2794      * negated scale, plus the number of characters in the converted
2795      * unscaled value, less one. That is,
2796      * {@code -scale+(ulength-1)}, where {@code ulength} is the
2797      * length of the absolute value of the unscaled value in decimal
2798      * digits (its <i>precision</i>).
2799      *
2800      * <p>If the scale is greater than or equal to zero and the
2801      * adjusted exponent is greater than or equal to {@code -6}, the
2802      * number will be converted to a character form without using
2803      * exponential notation. In this case, if the scale is zero then
2804      * no decimal point is added and if the scale is positive a
2805      * decimal point will be inserted with the scale specifying the
2806      * number of characters to the right of the decimal point.
2807      * {@code '0'} characters are added to the left of the converted

```

```

2808 * unscaled value as necessary. If no character precedes the
2809 * decimal point after this insertion then a conventional
2810 * {@code '0'} character is prefixed.
2811 *
2812 * <p>Otherwise (that is, if the scale is negative, or the
2813 * adjusted exponent is less than {@code -6}), the number will be
2814 * converted to a character form using exponential notation. In
2815 * this case, if the converted {@code BigInteger} has more than
2816 * one digit a decimal point is inserted after the first digit.
2817 * An exponent in character form is then suffixed to the converted
2818 * unscaled value (perhaps with inserted decimal point); this
2819 * comprises the letter {@code 'E'} followed immediately by the
2820 * adjusted exponent converted to a character form. The latter is
2821 * in base ten, using the characters {@code '0'} through
2822 * {@code '9'} with no leading zeros, and is always prefixed by a
2823 * sign character {@code '-'} (<tt>'&#92;u002D'</tt>) if the
2824 * adjusted exponent is negative, {@code '+'}
2825 * (<tt>'&#92;u002B'</tt>) otherwise).
2826 *
2827 * <p>Finally, the entire string is prefixed by a minus sign
2828 * character {@code '-'} (<tt>'&#92;u002D'</tt>) if the unscaled
2829 * value is less than zero. No sign character is prefixed if the
2830 * unscaled value is zero or positive.
2831 *
2832 * <p><b>Examples:</b>
2833 * <p>For each representation [<i>unscaled value</i>, <i>scale</i>]
2834 * on the left, the resulting string is shown on the right.
2835 * <pre>
2836 * [123,0]      "123"
2837 * [-123,0]     "-123"
2838 * [123,-1]     "1.23E+3"
2839 * [123,-3]     "1.23E+5"
2840 * [123,1]      "12.3"
2841 * [123,5]      "0.00123"
2842 * [123,10]     "1.23E-8"
2843 * [-123,12]    "-1.23E-10"
2844 * </pre>
2845 *
2846 * <b>Notes:</b>
2847 * <ol>
2848 *
2849 * <li>There is a one-to-one mapping between the distinguishable
2850 * {@code BigDecimal} values and the result of this conversion.
2851 * That is, every distinguishable {@code BigDecimal} value
2852 * (unscaled value and scale) has a unique string representation
2853 * as a result of using {@code toString}. If that string
2854 * representation is converted back to a {@code BigDecimal} using
2855 * the {@code link #BigDecimal(String)} constructor, then the original
2856 * value will be recovered.
2857 *
2858 * <li>The string produced for a given number is always the same;
2859 * it is not affected by locale. This means that it can be used
2860 * as a canonical string representation for exchanging decimal

```

```

2861 * data, or as a key for a Hashtable, etc. Locale-sensitive
2862 * number formatting and parsing is handled by the {@link
2863 * java.text.NumberFormat} class and its subclasses.
2864 *
2865 * <li>The {@link #toEngineeringString} method may be used for
2866 * presenting numbers with exponents in engineering notation, and the
2867 * {@link #setScale(int,RoundingMode) setScale} method may be used for
2868 * rounding a {@code BigDecimal} so it has a known number of digits after
2869 * the decimal point.
2870 *
2871 * <li>The digit-to-character mapping provided by
2872 * {@code Character.forDigit} is used.
2873 *
2874 * </ol>
2875 *
2876 * @return string representation of this {@code BigDecimal}.
2877 * @see Character#forDigit
2878 * @see #BigDecimal(java.lang.String)
2879 */
2880 @Override
2881 public String toString() {
2882     String sc = stringCache;
2883     if (sc == null)
2884         stringCache = sc = layoutChars(true);
2885     return sc;
2886 }
2887
2888 /**
2889 * Returns a string representation of this {@code BigDecimal},
2890 * using engineering notation if an exponent is needed.
2891 *
2892 * <p>Returns a string that represents the {@code BigDecimal} as
2893 * described in the {@link #toString()} method, except that if
2894 * exponential notation is used, the power of ten is adjusted to
2895 * be a multiple of three (engineering notation) such that the
2896 * integer part of nonzero values will be in the range 1 through
2897 * 999. If exponential notation is used for zero values, a
2898 * decimal point and one or two fractional zero digits are used so
2899 * that the scale of the zero value is preserved. Note that
2900 * unlike the output of {@link #toString()}, the output of this
2901 * method is not guaranteed to recover the same [integer,
2902 * scale] pair of this {@code BigDecimal} if the output string is
2903 * converting back to a {@code BigDecimal} using the {@linkplain
2904 * #BigDecimal(String) string constructor}. The result of this method meets
2905 * the weaker constraint of always producing a numerically equal
2906 * result from applying the string constructor to the method's output.
2907 *
2908 * @return string representation of this {@code BigDecimal}, using
2909 *         engineering notation if an exponent is needed.
2910 * @since 1.5
2911 */
2912 public String toEngineeringString() {
2913     return layoutChars(false);

```



```

2914     }
2915
2916     /**
2917      * Returns a string representation of this {@code BigDecimal}
2918      * without an exponent field. For values with a positive scale,
2919      * the number of digits to the right of the decimal point is used
2920      * to indicate scale. For values with a zero or negative scale,
2921      * the resulting string is generated as if the value were
2922      * converted to a numerically equal value with zero scale and as
2923      * if all the trailing zeros of the zero scale value were present
2924      * in the result.
2925      *
2926      * The entire string is prefixed by a minus sign character '-'
2927      * (<tt>'&#92;u002D'</tt>) if the unscaled value is less than
2928      * zero. No sign character is prefixed if the unscaled value is
2929      * zero or positive.
2930      *
2931      * Note that if the result of this method is passed to the
2932      * {@linkplain #BigDecimal(String) string constructor}, only the
2933      * numerical value of this {@code BigDecimal} will necessarily be
2934      * recovered; the representation of the new {@code BigDecimal}
2935      * may have a different scale. In particular, if this
2936      * {@code BigDecimal} has a negative scale, the string resulting
2937      * from this method will have a scale of zero when processed by
2938      * the string constructor.
2939      *
2940      * (This method behaves analogously to the {@code toString}
2941      * method in 1.4 and earlier releases.)
2942      *
2943      * @return a string representation of this {@code BigDecimal}
2944      * without an exponent field.
2945      * @since 1.5
2946      * @see #toString()
2947      * @see #toEngineeringString()
2948      */
2949     public String toPlainString() {
2950         if(scale==0) {
2951             if(intCompact!=INFLATED) {
2952                 return Long.toString(intCompact);
2953             } else {
2954                 return intVal.toString();
2955             }
2956         }
2957         if(this.scale<0) { // No decimal point
2958             if(signum()==0) {
2959                 return "0";
2960             }
2961             int tailingZeros = checkScaleNonZero((-long)scale);
2962             StringBuilder buf;
2963             if(intCompact!=INFLATED) {
2964                 buf = new StringBuilder(20+tailingZeros);
2965                 buf.append(intCompact);
2966             } else {

```



```

2967         String str = intVal.toString();
2968         buf = new StringBuilder(str.length()+tailingZeros);
2969         buf.append(str);
2970     }
2971     for (int i = 0; i < trailingZeros; i++)
2972         buf.append('0');
2973     return buf.toString();
2974 }
2975 String str ;
2976 if(intCompact!=INFLATED) {
2977     str = Long.toString(Math.abs(intCompact));
2978 } else {
2979     str = intVal.abs().toString();
2980 }
2981 return getValueString(signum(), str, scale);
2982 }
2983
2984 /* Returns a digit.digit string */
2985 private String getValueString(int signum, String intString, int scale) {
2986     /* Insert decimal point */
2987     StringBuilder buf;
2988     int insertionPoint = intString.length() - scale;
2989     if (insertionPoint == 0) { /* Point goes right before intVal */
2990         return (signum<0 ? "-0." : "0.") + intString;
2991     } else if (insertionPoint > 0) { /* Point goes inside intVal */
2992         buf = new StringBuilder(intString);
2993         buf.insert(insertionPoint, '.');
2994         if (signum < 0)
2995             buf.insert(0, '-');
2996     } else { /* We must insert zeros between point and intVal */
2997         buf = new StringBuilder(3-insertionPoint + intString.length());
2998         buf.append(signum<0 ? "-0." : "0.");
2999         for (int i=0; i<=insertionPoint; i++)
3000             buf.append('0');
3001         buf.append(intString);
3002     }
3003     return buf.toString();
3004 }
3005
3006 /**
3007  * Converts this {@code BigDecimal} to a {@code BigInteger}.
3008  * This conversion is analogous to the
3009  * <i>narrowing primitive conversion</i> from {@code double} to
3010  * {@code long} as defined in section 5.1.3 of
3011  * <cite>The Java&trade; Language Specification</cite>:
3012  * any fractional part of this
3013  * {@code BigDecimal} will be discarded. Note that this
3014  * conversion can lose information about the precision of the
3015  * {@code BigDecimal} value.
3016  * <p>
3017  * To have an exception thrown if the conversion is inexact (in
3018  * other words if a nonzero fractional part is discarded), use the
3019  * {@link #toBigIntegerExact()} method.

```

```

3020     *
3021     * @return this {@code BigDecimal} converted to a {@code BigInteger}.
3022     */
3023     public BigInteger toBigInteger() {
3024         // force to an integer, quietly
3025         return this.setScale(0, ROUND_DOWN).inflated();
3026     }
3027
3028     /**
3029     * Converts this {@code BigDecimal} to a {@code BigInteger},
3030     * checking for lost information. An exception is thrown if this
3031     * {@code BigDecimal} has a nonzero fractional part.
3032     *
3033     * @return this {@code BigDecimal} converted to a {@code BigInteger}.
3034     * @throws ArithmeticException if {@code this} has a nonzero
3035     *         fractional part.
3036     * @since 1.5
3037     */
3038     public BigInteger toBigIntegerExact() {
3039         // round to an integer, with Exception if decimal part non-0
3040         return this.setScale(0, ROUND_UNNECESSARY).inflated();
3041     }
3042
3043     /**
3044     * Converts this {@code BigDecimal} to a {@code long}.
3045     * This conversion is analogous to the
3046     * <i>narrowing primitive conversion</i> from {@code double} to
3047     * {@code short} as defined in section 5.1.3 of
3048     * <cite>The Java&trade; Language Specification</cite>:
3049     * any fractional part of this
3050     * {@code BigDecimal} will be discarded, and if the resulting
3051     * "{@code BigInteger}" is too big to fit in a
3052     * {@code long}, only the low-order 64 bits are returned.
3053     * Note that this conversion can lose information about the
3054     * overall magnitude and precision of this {@code BigDecimal} value as well
3055     * as return a result with the opposite sign.
3056     *
3057     * @return this {@code BigDecimal} converted to a {@code long}.
3058     */
3059     public long longValue(){
3060         return (intCompact != INFLATED && scale == 0) ?
3061             intCompact:
3062             toBigInteger().longValue();
3063     }
3064
3065     /**
3066     * Converts this {@code BigDecimal} to a {@code long}, checking
3067     * for lost information. If this {@code BigDecimal} has a
3068     * nonzero fractional part or is out of the possible range for a
3069     * {@code long} result then an {@code ArithmeticException} is
3070     * thrown.
3071     *
3072     * @return this {@code BigDecimal} converted to a {@code long}.

```

```

3073     * @throws ArithmeticException if {@code this} has a nonzero
3074     *     fractional part, or will not fit in a {@code long}.
3075     * @since 1.5
3076     */
3077     public long longValueExact() {
3078         if (intCompact != INFLATED && scale == 0)
3079             return intCompact;
3080         // If more than 19 digits in integer part it cannot possibly fit
3081         if ((precision() - scale) > 19) // [OK for negative scale too]
3082             throw new java.lang.ArithmeticException("Overflow");
3083         // Fastpath zero and < 1.0 numbers (the latter can be very slow
3084         // to round if very small)
3085         if (this.signum() == 0)
3086             return 0;
3087         if ((this.precision() - this.scale) <= 0)
3088             throw new ArithmeticException("Rounding necessary");
3089         // round to an integer, with Exception if decimal part non-0
3090         BigDecimal num = this.setScale(0, ROUND_UNNECESSARY);
3091         if (num.precision() >= 19) // need to check carefully
3092             LongOverflow.check(num);
3093         return num.inflated().longValue();
3094     }
3095
3096     private static class LongOverflow {
3097         /** BigInteger equal to Long.MIN_VALUE. */
3098         private static final BigInteger LONGMIN = BigInteger.valueOf(Long.MIN_VALUE);
3099
3100         /** BigInteger equal to Long.MAX_VALUE. */
3101         private static final BigInteger LONGMAX = BigInteger.valueOf(Long.MAX_VALUE);
3102
3103         public static void check(BigDecimal num) {
3104             BigInteger intVal = num.inflated();
3105             if (intVal.compareTo(LONGMIN) < 0 ||
3106                 intVal.compareTo(LONGMAX) > 0)
3107                 throw new java.lang.ArithmeticException("Overflow");
3108         }
3109     }
3110
3111     /**
3112     * Converts this {@code BigDecimal} to an {@code int}.
3113     * This conversion is analogous to the
3114     * <i>narrowing primitive conversion</i> from {@code double} to
3115     * {@code short} as defined in section 5.1.3 of
3116     * <cite>The Java&trade; Language Specification</cite>:
3117     * any fractional part of this
3118     * {@code BigDecimal} will be discarded, and if the resulting
3119     * "{@code BigInteger}" is too big to fit in an
3120     * {@code int}, only the low-order 32 bits are returned.
3121     * Note that this conversion can lose information about the
3122     * overall magnitude and precision of this {@code BigDecimal}
3123     * value as well as return a result with the opposite sign.
3124     *
3125     * @return this {@code BigDecimal} converted to an {@code int}.

```

```

3126     */
3127     public int intValue() {
3128         return (intCompact != INFLATED && scale == 0) ?
3129             (int)intCompact :
3130             toBigInteger().intValue();
3131     }
3132
3133     /**
3134      * Converts this {@code BigDecimal} to an {@code int}, checking
3135      * for lost information. If this {@code BigDecimal} has a
3136      * nonzero fractional part or is out of the possible range for an
3137      * {@code int} result then an {@code ArithmeticException} is
3138      * thrown.
3139      *
3140      * @return this {@code BigDecimal} converted to an {@code int}.
3141      * @throws ArithmeticException if {@code this} has a nonzero
3142      *         fractional part, or will not fit in an {@code int}.
3143      * @since 1.5
3144      */
3145     public int intValueExact() {
3146         long num;
3147         num = this.longValueExact();    // will check decimal part
3148         if ((int)num != num)
3149             throw new java.lang.ArithmeticException("Overflow");
3150         return (int)num;
3151     }
3152
3153     /**
3154      * Converts this {@code BigDecimal} to a {@code short}, checking
3155      * for lost information. If this {@code BigDecimal} has a
3156      * nonzero fractional part or is out of the possible range for a
3157      * {@code short} result then an {@code ArithmeticException} is
3158      * thrown.
3159      *
3160      * @return this {@code BigDecimal} converted to a {@code short}.
3161      * @throws ArithmeticException if {@code this} has a nonzero
3162      *         fractional part, or will not fit in a {@code short}.
3163      * @since 1.5
3164      */
3165     public short shortValueExact() {
3166         long num;
3167         num = this.longValueExact();    // will check decimal part
3168         if ((short)num != num)
3169             throw new java.lang.ArithmeticException("Overflow");
3170         return (short)num;
3171     }
3172
3173     /**
3174      * Converts this {@code BigDecimal} to a {@code byte}, checking
3175      * for lost information. If this {@code BigDecimal} has a
3176      * nonzero fractional part or is out of the possible range for a
3177      * {@code byte} result then an {@code ArithmeticException} is
3178      * thrown.

```

```

3179      *
3180      * @return this {@code BigDecimal} converted to a {@code byte}.
3181      * @throws ArithmeticException if {@code this} has a nonzero
3182      *         fractional part, or will not fit in a {@code byte}.
3183      * @since 1.5
3184      */
3185      public byte byteValueExact() {
3186          long num;
3187          num = this.longValueExact();    // will check decimal part
3188          if ((byte)num != num)
3189              throw new java.lang.ArithmeticException("Overflow");
3190          return (byte)num;
3191      }
3192
3193      /**
3194      * Converts this {@code BigDecimal} to a {@code float}.
3195      * This conversion is similar to the
3196      * <i>narrowing primitive conversion</i> from {@code double} to
3197      * {@code float} as defined in section 5.1.3 of
3198      * <cite>The Java&trade; Language Specification</cite>:
3199      * if this {@code BigDecimal} has too great a
3200      * magnitude to represent as a {@code float}, it will be
3201      * converted to {@link Float#NEGATIVE_INFINITY} or {@link
3202      * Float#POSITIVE_INFINITY} as appropriate. Note that even when
3203      * the return value is finite, this conversion can lose
3204      * information about the precision of the {@code BigDecimal}
3205      * value.
3206      *
3207      * @return this {@code BigDecimal} converted to a {@code float}.
3208      */
3209      public float floatValue(){
3210          if(intCompact != INFLATED) {
3211              if (scale == 0) {
3212                  return (float)intCompact;
3213              } else {
3214                  /*
3215                   * If both intCompact and the scale can be exactly
3216                   * represented as float values, perform a single float
3217                   * multiply or divide to compute the (properly
3218                   * rounded) result.
3219                   */
3220                  if (Math.abs(intCompact) < 1L<<22 ) {
3221                      // Don't have too guard against
3222                      // Math.abs(MIN_VALUE) because of outer check
3223                      // against INFLATED.
3224                      if (scale > 0 && scale < float10pow.length) {
3225                          return (float)intCompact / float10pow[scale];
3226                      } else if (scale < 0 && scale > -float10pow.length) {
3227                          return (float)intCompact * float10pow[-scale];
3228                      }
3229                  }
3230              }
3231          }

```

```

3232     // Somewhat inefficient, but guaranteed to work.
3233     return Float.parseFloat(this.toString());
3234 }
3235
3236 /**
3237  * Converts this {@code BigDecimal} to a {@code double}.
3238  * This conversion is similar to the
3239  * <i>narrowing primitive conversion</i> from {@code double} to
3240  * {@code float} as defined in section 5.1.3 of
3241  * <cite>The Java&trade; Language Specification</cite>:
3242  * if this {@code BigDecimal} has too great a
3243  * magnitude represent as a {@code double}, it will be
3244  * converted to {@link Double#NEGATIVE_INFINITY} or {@link
3245  * Double#POSITIVE_INFINITY} as appropriate. Note that even when
3246  * the return value is finite, this conversion can lose
3247  * information about the precision of the {@code BigDecimal}
3248  * value.
3249  *
3250  * @return this {@code BigDecimal} converted to a {@code double}.
3251  */
3252 public double doubleValue(){
3253     if(intCompact != INFLATED) {
3254         if (scale == 0) {
3255             return (double)intCompact;
3256         } else {
3257             /*
3258              * If both intCompact and the scale can be exactly
3259              * represented as double values, perform a single
3260              * double multiply or divide to compute the (properly
3261              * rounded) result.
3262              */
3263             if (Math.abs(intCompact) < 1L<<52 ) {
3264                 // Don't have to guard against
3265                 // Math.abs(MIN_VALUE) because of outer check
3266                 // against INFLATED.
3267                 if (scale > 0 && scale < double10pow.length) {
3268                     return (double)intCompact / double10pow[scale];
3269                 } else if (scale < 0 && scale > -double10pow.length) {
3270                     return (double)intCompact * double10pow[-scale];
3271                 }
3272             }
3273         }
3274     }
3275     // Somewhat inefficient, but guaranteed to work.
3276     return Double.parseDouble(this.toString());
3277 }
3278
3279 /**
3280  * Powers of 10 which can be represented exactly in {@code
3281  * double}.
3282  */
3283 private static final double double10pow[] = {
3284     1.0e0, 1.0e1, 1.0e2, 1.0e3, 1.0e4, 1.0e5,

```

```

3285     1.0e6, 1.0e7, 1.0e8, 1.0e9, 1.0e10, 1.0e11,
3286     1.0e12, 1.0e13, 1.0e14, 1.0e15, 1.0e16, 1.0e17,
3287     1.0e18, 1.0e19, 1.0e20, 1.0e21, 1.0e22
3288 };
3289
3290 /**
3291  * Powers of 10 which can be represented exactly in {@code
3292  * float}.
3293  */
3294 private static final float float10pow[] = {
3295     1.0e0f, 1.0e1f, 1.0e2f, 1.0e3f, 1.0e4f, 1.0e5f,
3296     1.0e6f, 1.0e7f, 1.0e8f, 1.0e9f, 1.0e10f
3297 };
3298
3299 /**
3300  * Returns the size of an ulp, a unit in the last place, of this
3301  * {@code BigDecimal}. An ulp of a nonzero {@code BigDecimal}
3302  * value is the positive distance between this value and the
3303  * {@code BigDecimal} value next larger in magnitude with the
3304  * same number of digits. An ulp of a zero value is numerically
3305  * equal to 1 with the scale of {@code this}. The result is
3306  * stored with the same scale as {@code this} so the result
3307  * for zero and nonzero values is equal to {@code [1,
3308  * this.scale()]}].
3309  *
3310  * @return the size of an ulp of {@code this}
3311  * @since 1.5
3312  */
3313 public BigDecimal ulp() {
3314     return BigDecimal.valueOf(1, this.scale(), 1);
3315 }
3316
3317 // Private class to build a string representation for BigDecimal object.
3318 // "StringBuilderHelper" is constructed as a thread local variable so it is
3319 // thread safe. The StringBuilder field acts as a buffer to hold the temporary
3320 // representation of BigDecimal. The cmpCharArray holds all the characters for
3321 // the compact representation of BigDecimal (except for '-' sign' if it is
3322 // negative) if its intCompact field is not INFLATED. It is shared by all
3323 // calls to toString() and its variants in that particular thread.
3324 static class StringBuilderHelper {
3325     final StringBuilder sb; // Placeholder for BigDecimal string
3326     final char[] cmpCharArray; // character array to place the intCompact
3327
3328     StringBuilderHelper() {
3329         sb = new StringBuilder();
3330         // All non negative longs can be made to fit into 19 character array.
3331         cmpCharArray = new char[19];
3332     }
3333
3334     // Accessors.
3335     StringBuilder getStringBuilder() {
3336         sb.setLength(0);
3337         return sb;

```

```

3338     }
3339
3340     char[] getCompactCharArray() {
3341         return cmpCharArray;
3342     }
3343
3344     /**
3345      * Places characters representing the intCompact in {@code long} into
3346      * cmpCharArray and returns the offset to the array where the
3347      * representation starts.
3348      *
3349      * @param intCompact the number to put into the cmpCharArray.
3350      * @return offset to the array where the representation starts.
3351      * Note: intCompact must be greater or equal to zero.
3352      */
3353     int putIntCompact(long intCompact) {
3354         assert intCompact >= 0;
3355
3356         long q;
3357         int r;
3358         // since we start from the least significant digit, charPos points to
3359         // the last character in cmpCharArray.
3360         int charPos = cmpCharArray.length;
3361
3362         // Get 2 digits/iteration using longs until quotient fits into an int
3363         while (intCompact > Integer.MAX_VALUE) {
3364             q = intCompact / 100;
3365             r = (int)(intCompact - q * 100);
3366             intCompact = q;
3367             cmpCharArray[--charPos] = DIGIT_ONES[r];
3368             cmpCharArray[--charPos] = DIGIT_TENS[r];
3369         }
3370
3371         // Get 2 digits/iteration using ints when i2 >= 100
3372         int q2;
3373         int i2 = (int)intCompact;
3374         while (i2 >= 100) {
3375             q2 = i2 / 100;
3376             r = i2 - q2 * 100;
3377             i2 = q2;
3378             cmpCharArray[--charPos] = DIGIT_ONES[r];
3379             cmpCharArray[--charPos] = DIGIT_TENS[r];
3380         }
3381
3382         cmpCharArray[--charPos] = DIGIT_ONES[i2];
3383         if (i2 >= 10)
3384             cmpCharArray[--charPos] = DIGIT_TENS[i2];
3385
3386         return charPos;
3387     }
3388
3389     final static char[] DIGIT_TENS = {
3390         '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',

```



```

3391         '1', '1', '1', '1', '1', '1', '1', '1', '1', '1',
3392         '2', '2', '2', '2', '2', '2', '2', '2', '2', '2',
3393         '3', '3', '3', '3', '3', '3', '3', '3', '3', '3',
3394         '4', '4', '4', '4', '4', '4', '4', '4', '4', '4',
3395         '5', '5', '5', '5', '5', '5', '5', '5', '5', '5',
3396         '6', '6', '6', '6', '6', '6', '6', '6', '6', '6',
3397         '7', '7', '7', '7', '7', '7', '7', '7', '7', '7',
3398         '8', '8', '8', '8', '8', '8', '8', '8', '8', '8',
3399         '9', '9', '9', '9', '9', '9', '9', '9', '9', '9',
3400     };
3401
3402     final static char[] DIGIT_ONES = {
3403         '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
3404         '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
3405         '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
3406         '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
3407         '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
3408         '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
3409         '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
3410         '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
3411         '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
3412         '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
3413     };
3414 }
3415
3416 /**
3417  * Lay out this {@code BigDecimal} into a {@code char[]} array.
3418  * The Java 1.2 equivalent to this was called {@code getValueString}.
3419  *
3420  * @param sci {@code true} for Scientific exponential notation;
3421  *           {@code false} for Engineering
3422  * @return string with canonical string representation of this
3423  *         {@code BigDecimal}
3424  */
3425 private String layoutChars(boolean sci) {
3426     if (scale == 0) // zero scale is trivial
3427         return (intCompact != INFLATED) ?
3428             Long.toString(intCompact):
3429             intVal.toString();
3430     if (scale == 2 &&
3431         intCompact >= 0 && intCompact < Integer.MAX_VALUE) {
3432         // currency fast path
3433         int lowInt = (int)intCompact % 100;
3434         int highInt = (int)intCompact / 100;
3435         return (Integer.toString(highInt) + '.' +
3436             StringBuilderHelper.DIGIT_TENS[lowInt] +
3437             StringBuilderHelper.DIGIT_ONES[lowInt]) ;
3438     }
3439
3440     StringBuilderHelper sbHelper = threadLocalStringBuilderHelper.get();
3441     char[] coeff;
3442     int offset; // offset is the starting index for coeff array
3443     // Get the significand as an absolute value

```

```

3444     if (intCompact != INFLATED) {
3445         offset = sbHelper.putIntCompact(Math.abs(intCompact));
3446         coeff = sbHelper.getCompactCharArray();
3447     } else {
3448         offset = 0;
3449         coeff = intVal.abs().toString().toCharArray();
3450     }
3451
3452     // Construct a buffer, with sufficient capacity for all cases.
3453     // If E-notation is needed, length will be: +1 if negative, +1
3454     // if '.' needed, +2 for "E+", + up to 10 for adjusted exponent.
3455     // Otherwise it could have +1 if negative, plus leading "0.00000"
3456     StringBuilder buf = sbHelper.getStringBuilder();
3457     if (signum() < 0)           // prefix '-' if negative
3458         buf.append('-');
3459     int coeffLen = coeff.length - offset;
3460     long adjusted = -(long)scale + (coeffLen - 1);
3461     if ((scale >= 0) && (adjusted >= -6)) { // plain number
3462         int pad = scale - coeffLen;        // count of padding zeros
3463         if (pad >= 0) {                    // 0.xxx form
3464             buf.append('0');
3465             buf.append('.');
3466             for (; pad > 0; pad--) {
3467                 buf.append('0');
3468             }
3469             buf.append(coeff, offset, coeffLen);
3470         } else {                          // xx.xx form
3471             buf.append(coeff, offset, -pad);
3472             buf.append('.');
3473             buf.append(coeff, -pad + offset, scale);
3474         }
3475     } else { // E-notation is needed
3476         if (sci) {                        // Scientific notation
3477             buf.append(coeff[offset]);    // first character
3478             if (coeffLen > 1) {           // more to come
3479                 buf.append('.');
3480                 buf.append(coeff, offset + 1, coeffLen - 1);
3481             }
3482         } else {                          // Engineering notation
3483             int sig = (int)(adjusted % 3);
3484             if (sig < 0)
3485                 sig += 3;                 // [adjusted was negative]
3486             adjusted -= sig;              // now a multiple of 3
3487             sig++;
3488             if (signum() == 0) {
3489                 switch (sig) {
3490                     case 1:
3491                         buf.append('0'); // exponent is a multiple of three
3492                         break;
3493                     case 2:
3494                         buf.append("0.00");
3495                         adjusted += 3;
3496                         break;

```

```

3497         case 3:
3498             buf.append("0.0");
3499             adjusted += 3;
3500             break;
3501         default:
3502             throw new AssertionError("Unexpected sig value " + sig);
3503     }
3504     } else if (sig >= coeffLen) { // significand all in integer
3505         buf.append(coeff, offset, coeffLen);
3506         // may need some zeros, too
3507         for (int i = sig - coeffLen; i > 0; i--)
3508             buf.append('0');
3509     } else { // xx.xxE form
3510         buf.append(coeff, offset, sig);
3511         buf.append('.');
3512         buf.append(coeff, offset + sig, coeffLen - sig);
3513     }
3514 }
3515 if (adjusted != 0) { // [!sci could have made 0]
3516     buf.append('E');
3517     if (adjusted > 0) // force sign for positive
3518         buf.append('+');
3519     buf.append(adjusted);
3520 }
3521 }
3522 return buf.toString();
3523 }
3524
3525 /**
3526  * Return 10 to the power n, as a {@code BigInteger}.
3527  *
3528  * @param n the power of ten to be returned (>=0)
3529  * @return a {@code BigInteger} with the value  $10^n$ 
3530  */
3531 private static BigInteger bigTenToThe(int n) {
3532     if (n < 0)
3533         return BigInteger.ZERO;
3534
3535     if (n < BIG_TEN_POWERS_TABLE_MAX) {
3536         BigInteger[] pows = BIG_TEN_POWERS_TABLE;
3537         if (n < pows.length)
3538             return pows[n];
3539         else
3540             return expandBigIntegerTenPowers(n);
3541     }
3542
3543     return BigInteger.TEN.pow(n);
3544 }
3545
3546 /**
3547  * Expand the BIG_TEN_POWERS_TABLE array to contain at least  $10^n$ .
3548  *
3549  * @param n the power of ten to be returned (>=0)

```

```

3550     * @return a {@code BigDecimal} with the value  $10^n$  and
3551     *         in the meantime, the BIG_TEN_POWERS_TABLE array gets
3552     *         expanded to the size greater than n.
3553     */
3554     private static BigInteger expandBigIntegerTenPowers(int n) {
3555         synchronized(BigDecimal.class) {
3556             BigInteger[] pows = BIG_TEN_POWERS_TABLE;
3557             int curLen = pows.length;
3558             // The following comparison and the above synchronized statement is
3559             // to prevent multiple threads from expanding the same array.
3560             if (curLen <= n) {
3561                 int newLen = curLen << 1;
3562                 while (newLen <= n)
3563                     newLen <<= 1;
3564                 pows = Arrays.copyOf(pows, newLen);
3565                 for (int i = curLen; i < newLen; i++)
3566                     pows[i] = pows[i - 1].multiply(BigInteger.TEN);
3567                 // Based on the following facts:
3568                 // 1. pows is a private local variable;
3569                 // 2. the following store is a volatile store.
3570                 // the newly created array elements can be safely published.
3571                 BIG_TEN_POWERS_TABLE = pows;
3572             }
3573             return pows[n];
3574         }
3575     }
3576
3577     private static final long[] LONG_TEN_POWERS_TABLE = {
3578         1, // 0 / 10^0
3579         10, // 1 / 10^1
3580         100, // 2 / 10^2
3581         1000, // 3 / 10^3
3582         10000, // 4 / 10^4
3583         100000, // 5 / 10^5
3584         1000000, // 6 / 10^6
3585         10000000, // 7 / 10^7
3586         100000000, // 8 / 10^8
3587         1000000000, // 9 / 10^9
3588         10000000000L, // 10 / 10^10
3589         100000000000L, // 11 / 10^11
3590         1000000000000L, // 12 / 10^12
3591         10000000000000L, // 13 / 10^13
3592         100000000000000L, // 14 / 10^14
3593         1000000000000000L, // 15 / 10^15
3594         10000000000000000L, // 16 / 10^16
3595         100000000000000000L, // 17 / 10^17
3596         1000000000000000000L, // 18 / 10^18
3597     };
3598
3599     private static volatile BigInteger BIG_TEN_POWERS_TABLE[] = {
3600         BigInteger.ONE,
3601         BigInteger.valueOf(10),
3602         BigInteger.valueOf(100),

```

```

3603     BigInteger.valueOf(1000),
3604     BigInteger.valueOf(10000),
3605     BigInteger.valueOf(100000),
3606     BigInteger.valueOf(1000000),
3607     BigInteger.valueOf(10000000),
3608     BigInteger.valueOf(100000000),
3609     BigInteger.valueOf(1000000000),
3610     BigInteger.valueOf(10000000000L),
3611     BigInteger.valueOf(100000000000L),
3612     BigInteger.valueOf(1000000000000L),
3613     BigInteger.valueOf(10000000000000L),
3614     BigInteger.valueOf(100000000000000L),
3615     BigInteger.valueOf(1000000000000000L),
3616     BigInteger.valueOf(10000000000000000L),
3617     BigInteger.valueOf(100000000000000000L),
3618     BigInteger.valueOf(1000000000000000000L)
3619 };
3620
3621 private static final int BIG_TEN_POWERS_TABLE_INITLEN =
3622     BIG_TEN_POWERS_TABLE.length;
3623 private static final int BIG_TEN_POWERS_TABLE_MAX =
3624     16 * BIG_TEN_POWERS_TABLE_INITLEN;
3625
3626 private static final long THRESHOLDS_TABLE[] = {
3627     Long.MAX_VALUE,           // 0
3628     Long.MAX_VALUE/10L,       // 1
3629     Long.MAX_VALUE/100L,      // 2
3630     Long.MAX_VALUE/1000L,     // 3
3631     Long.MAX_VALUE/10000L,    // 4
3632     Long.MAX_VALUE/100000L,   // 5
3633     Long.MAX_VALUE/1000000L,  // 6
3634     Long.MAX_VALUE/10000000L, // 7
3635     Long.MAX_VALUE/100000000L, // 8
3636     Long.MAX_VALUE/1000000000L, // 9
3637     Long.MAX_VALUE/10000000000L, // 10
3638     Long.MAX_VALUE/100000000000L, // 11
3639     Long.MAX_VALUE/1000000000000L, // 12
3640     Long.MAX_VALUE/10000000000000L, // 13
3641     Long.MAX_VALUE/100000000000000L, // 14
3642     Long.MAX_VALUE/1000000000000000L, // 15
3643     Long.MAX_VALUE/10000000000000000L, // 16
3644     Long.MAX_VALUE/100000000000000000L, // 17
3645     Long.MAX_VALUE/1000000000000000000L // 18
3646 };
3647
3648 /**
3649  * Compute val * 10 ^ n; return this product if it is
3650  * representable as a long, INFLATED otherwise.
3651  */
3652 private static long longMultiplyPowerTen(long val, int n) {
3653     if (val == 0 || n <= 0)
3654         return val;
3655     long[] tab = LONG_TEN_POWERS_TABLE;

```

```

3656     long[] bounds = THRESHOLDS_TABLE;
3657     if (n < tab.length && n < bounds.length) {
3658         long tenpower = tab[n];
3659         if (val == 1)
3660             return tenpower;
3661         if (Math.abs(val) <= bounds[n])
3662             return val * tenpower;
3663     }
3664     return INFLATED;
3665 }
3666
3667 /**
3668  * Compute this * 10 ^ n.
3669  * Needed mainly to allow special casing to trap zero value
3670  */
3671 private BigInteger bigMultiplyPowerTen(int n) {
3672     if (n <= 0)
3673         return this.inflated();
3674
3675     if (intCompact != INFLATED)
3676         return bigTenToThe(n).multiply(intCompact);
3677     else
3678         return intVal.multiply(bigTenToThe(n));
3679 }
3680
3681 /**
3682  * Returns appropriate BigInteger from intVal field if intVal is
3683  * null, i.e. the compact representation is in use.
3684  */
3685 private BigInteger inflated() {
3686     if (intVal == null) {
3687         return BigInteger.valueOf(intCompact);
3688     }
3689     return intVal;
3690 }
3691
3692 /**
3693  * Match the scales of two {@code BigDecimal}s to align their
3694  * least significant digits.
3695  *
3696  * <p>If the scales of val[0] and val[1] differ, rescale
3697  * (non-destructively) the lower-scaled {@code BigDecimal} so
3698  * they match. That is, the lower-scaled reference will be
3699  * replaced by a reference to a new object with the same scale as
3700  * the other {@code BigDecimal}.
3701  *
3702  * @param val array of two elements referring to the two
3703  *             {@code BigDecimal}s to be aligned.
3704  */
3705 private static void matchScale(BigDecimal[] val) {
3706     if (val[0].scale == val[1].scale) {
3707         return;
3708     } else if (val[0].scale < val[1].scale) {

```

```

3709         val[0] = val[0].setScale(val[1].scale, ROUND_UNNECESSARY);
3710     } else if (val[1].scale < val[0].scale) {
3711         val[1] = val[1].setScale(val[0].scale, ROUND_UNNECESSARY);
3712     }
3713 }
3714
3715 private static class UnsafeHolder {
3716     private static final sun.misc.Unsafe unsafe;
3717     private static final long intCompactOffset;
3718     private static final long intValOffset;
3719     static {
3720         try {
3721             unsafe = sun.misc.Unsafe.getUnsafe();
3722             intCompactOffset = unsafe.objectFieldOffset
3723                 (BigDecimal.class.getDeclaredField("intCompact"));
3724             intValOffset = unsafe.objectFieldOffset
3725                 (BigDecimal.class.getDeclaredField("intVal"));
3726         } catch (Exception ex) {
3727             throw new ExceptionInInitializerError(ex);
3728         }
3729     }
3730     static void setIntCompactVolatile(BigDecimal bd, long val) {
3731         unsafe.putLongVolatile(bd, intCompactOffset, val);
3732     }
3733
3734     static void setIntValVolatile(BigDecimal bd, BigInteger val) {
3735         unsafe.putObjectVolatile(bd, intValOffset, val);
3736     }
3737 }
3738
3739 /**
3740  * Reconstitute the {@code BigDecimal} instance from a stream (that is,
3741  * deserialize it).
3742  *
3743  * @param s the stream being read.
3744  */
3745 private void readObject(java.io.ObjectInputStream s)
3746     throws java.io.IOException, ClassNotFoundException {
3747     // Read in all fields
3748     s.defaultReadObject();
3749     // validate possibly bad fields
3750     if (intVal == null) {
3751         String message = "BigDecimal: null intVal in stream";
3752         throw new java.io.StreamCorruptedException(message);
3753     } // [all values of scale are now allowed]
3754     }
3755     UnsafeHolder.setIntCompactVolatile(this, compactValFor(intVal));
3756 }
3757
3758 /**
3759  * Serialize this {@code BigDecimal} to the stream in question
3760  *
3761  * @param s the stream to serialize to.

```

```

3762 */
3763 private void writeObject(java.io.ObjectOutputStream s)
3764     throws java.io.IOException {
3765     // Must inflate to maintain compatible serial form.
3766     if (this.intVal == null)
3767         UnsafeHolder.setIntValVolatile(this, BigInteger.valueOf(this.intCompact));
3768     // Could reset intVal back to null if it has to be set.
3769     s.defaultWriteObject();
3770 }
3771
3772 /**
3773  * Returns the length of the absolute value of a {@code long}, in decimal
3774  * digits.
3775  *
3776  * @param x the {@code long}
3777  * @return the length of the unscaled value, in decimal digits.
3778  */
3779 static int longDigitLength(long x) {
3780     /*
3781      * As described in "Bit Twiddling Hacks" by Sean Anderson,
3782      * (http://graphics.stanford.edu/~seander/bithacks.html)
3783      * integer log 10 of x is within 1 of (1233/4096)* (1 +
3784      * integer log 2 of x). The fraction 1233/4096 approximates
3785      * log10(2). So we first do a version of log2 (a variant of
3786      * Long class with pre-checks and opposite directionality) and
3787      * then scale and check against powers table. This is a little
3788      * simpler in present context than the version in Hacker's
3789      * Delight sec 11-4. Adding one to bit length allows comparing
3790      * downward from the LONG_TEN_POWERS_TABLE that we need
3791      * anyway.
3792      */
3793     assert x != BigDecimal.INFLATED;
3794     if (x < 0)
3795         x = -x;
3796     if (x < 10) // must screen for 0, might as well 10
3797         return 1;
3798     int r = ((64 - Long.numberOfLeadingZeros(x) + 1) * 1233) >>> 12;
3799     long[] tab = LONG_TEN_POWERS_TABLE;
3800     // if r >= length, must have max possible digits for long
3801     return (r >= tab.length || x < tab[r]) ? r : r + 1;
3802 }
3803
3804 /**
3805  * Returns the length of the absolute value of a BigInteger, in
3806  * decimal digits.
3807  *
3808  * @param b the BigInteger
3809  * @return the length of the unscaled value, in decimal digits
3810  */
3811 private static int bigDigitLength(BigInteger b) {
3812     /*
3813      * Same idea as the long version, but we need a better
3814      * approximation of log10(2). Using 646456993/2^31

```



```

3815     * is accurate up to max possible reported bitLength.
3816     */
3817     if (b.signum == 0)
3818         return 1;
3819     int r = (int)((((long)b.bitLength() + 1) * 646456993) >>> 31);
3820     return b.compareMagnitude(bigTenToThe(r)) < 0? r : r+1;
3821 }
3822
3823 /**
3824  * Check a scale for Underflow or Overflow. If this BigDecimal is
3825  * nonzero, throw an exception if the scale is outof range. If this
3826  * is zero, saturate the scale to the extreme value of the right
3827  * sign if the scale is out of range.
3828  *
3829  * @param val The new scale.
3830  * @throws ArithmeticException (overflow or underflow) if the new
3831  *         scale is out of range.
3832  * @return validated scale as an int.
3833  */
3834 private int checkScale(long val) {
3835     int asInt = (int)val;
3836     if (asInt != val) {
3837         asInt = val > Integer.MAX_VALUE ? Integer.MAX_VALUE : Integer.MIN_VALUE;
3838         BigInteger b;
3839         if (intCompact != 0 &&
3840             ((b = intVal) == null || b.signum() != 0))
3841             throw new ArithmeticException(asInt > 0 ? "Underflow" : "Overflow");
3842     }
3843     return asInt;
3844 }
3845
3846 /**
3847  * Returns the compact value for given {@code BigInteger}, or
3848  * INFLATED if too big. Relies on internal representation of
3849  * {@code BigInteger}.
3850  */
3851 private static long compactValFor(BigInteger b) {
3852     int[] m = b.mag;
3853     int len = m.length;
3854     if (len == 0)
3855         return 0;
3856     int d = m[0];
3857     if (len > 2 || (len == 2 && d < 0))
3858         return INFLATED;
3859
3860     long u = (len == 2)?
3861         (((long) m[1] & LONG_MASK) + (((long)d) << 32)) :
3862         (((long)d) & LONG_MASK);
3863     return (b.signum < 0)? -u : u;
3864 }
3865
3866 private static int longCompareMagnitude(long x, long y) {
3867     if (x < 0)

```

```

3868         x = -x;
3869         if (y < 0)
3870             y = -y;
3871         return (x < y) ? -1 : ((x == y) ? 0 : 1);
3872     }
3873
3874     private static int saturateLong(long s) {
3875         int i = (int)s;
3876         return (s == i) ? i : (s < 0 ? Integer.MIN_VALUE : Integer.MAX_VALUE);
3877     }
3878
3879     /*
3880     * Internal printing routine
3881     */
3882     private static void print(String name, BigDecimal bd) {
3883         System.err.format("%s:\tintCompact %d\tintVal %d\tscale %d\tprecision %d%n",
3884             name,
3885             bd.intCompact,
3886             bd.intVal,
3887             bd.scale,
3888             bd.precision);
3889     }
3890
3891     /**
3892     * Check internal invariants of this BigDecimal. These invariants
3893     * include:
3894     *
3895     * <ul>
3896     *
3897     * <li>The object must be initialized; either intCompact must not be
3898     * INFLATED or intVal is non-null. Both of these conditions may
3899     * be true.
3900     *
3901     * <li>If both intCompact and intVal and set, their values must be
3902     * consistent.
3903     *
3904     * <li>If precision is nonzero, it must have the right value.
3905     * </ul>
3906     *
3907     * Note: Since this is an audit method, we are not supposed to change the
3908     * state of this BigDecimal object.
3909     */
3910     private BigDecimal audit() {
3911         if (intCompact == INFLATED) {
3912             if (intVal == null) {
3913                 print("audit", this);
3914                 throw new AssertionError("null intVal");
3915             }
3916             // Check precision
3917             if (precision > 0 && precision != bigDigitLength(intVal)) {
3918                 print("audit", this);
3919                 throw new AssertionError("precision mismatch");
3920             }

```

```

3921     } else {
3922         if (intVal != null) {
3923             long val = intVal.longValue();
3924             if (val != intCompact) {
3925                 print("audit", this);
3926                 throw new AssertionError("Inconsistent state, intCompact=" +
3927                                         intCompact + "\t intVal=" + val);
3928             }
3929         }
3930         // Check precision
3931         if (precision > 0 && precision != longDigitLength(intCompact)) {
3932             print("audit", this);
3933             throw new AssertionError("precision mismatch");
3934         }
3935     }
3936     return this;
3937 }
3938
3939 /* the same as checkScale where value!=0 */
3940 private static int checkScaleNonZero(long val) {
3941     int asInt = (int)val;
3942     if (asInt != val) {
3943         throw new ArithmeticException(asInt>0 ? "Underflow":"Overflow");
3944     }
3945     return asInt;
3946 }
3947
3948 private static int checkScale(long intCompact, long val) {
3949     int asInt = (int)val;
3950     if (asInt != val) {
3951         asInt = val>Integer.MAX_VALUE ? Integer.MAX_VALUE : Integer.MIN_VALUE;
3952         if (intCompact != 0)
3953             throw new ArithmeticException(asInt>0 ? "Underflow":"Overflow");
3954     }
3955     return asInt;
3956 }
3957
3958 private static int checkScale(BigInteger intVal, long val) {
3959     int asInt = (int)val;
3960     if (asInt != val) {
3961         asInt = val>Integer.MAX_VALUE ? Integer.MAX_VALUE : Integer.MIN_VALUE;
3962         if (intVal.signum() != 0)
3963             throw new ArithmeticException(asInt>0 ? "Underflow":"Overflow");
3964     }
3965     return asInt;
3966 }
3967
3968 /**
3969  * Returns a {@code BigDecimal} rounded according to the MathContext
3970  * settings;
3971  * If rounding is needed a new {@code BigDecimal} is created and returned.
3972  *
3973  * @param val the value to be rounded

```

```

3974     * @param mc the context to use.
3975     * @return a {@code BigDecimal} rounded according to the MathContext
3976     *         settings. May return {@code value}, if no rounding needed.
3977     * @throws ArithmeticException if the rounding mode is
3978     *         {@code RoundingMode.UNNECESSARY} and the
3979     *         result is inexact.
3980     */
3981     private static BigDecimal doRound(BigDecimal val, MathContext mc) {
3982         int mcp = mc.precision;
3983         boolean wasDivided = false;
3984         if (mcp > 0) {
3985             BigInteger intVal = val.intVal;
3986             long compactVal = val.intCompact;
3987             int scale = val.scale;
3988             int prec = val.precision();
3989             int mode = mc.roundingMode.oldMode;
3990             int drop;
3991             if (compactVal == INFLATED) {
3992                 drop = prec - mcp;
3993                 while (drop > 0) {
3994                     scale = checkScaleNonZero((long) scale - drop);
3995                     intVal = divideAndRoundByTenPow(intVal, drop, mode);
3996                     wasDivided = true;
3997                     compactVal = compactValFor(intVal);
3998                     if (compactVal != INFLATED) {
3999                         prec = longDigitLength(compactVal);
4000                         break;
4001                     }
4002                     prec = bigDigitLength(intVal);
4003                     drop = prec - mcp;
4004                 }
4005             }
4006             if (compactVal != INFLATED) {
4007                 drop = prec - mcp; // drop can't be more than 18
4008                 while (drop > 0) {
4009                     scale = checkScaleNonZero((long) scale - drop);
4010                     compactVal = divideAndRound(compactVal, LONG_TEN_POWERS_TABLE[drop], mc.
4011                                             roundingMode.oldMode);
4012                     wasDivided = true;
4013                     prec = longDigitLength(compactVal);
4014                     drop = prec - mcp;
4015                     intVal = null;
4016                 }
4017             }
4018             return wasDivided ? new BigDecimal(intVal, compactVal, scale, prec) : val;
4019         }
4020         return val;
4021     }
4022     /*
4023     * Returns a {@code BigDecimal} created from {@code long} value with
4024     * given scale rounded according to the MathContext settings
4025     */

```

```

4026     private static BigDecimal doRound(long compactVal, int scale, MathContext mc) {
4027         int mcp = mc.precision;
4028         if (mcp > 0 && mcp < 19) {
4029             int prec = longDigitLength(compactVal);
4030             int drop = prec - mcp; // drop can't be more than 18
4031             while (drop > 0) {
4032                 scale = checkScaleNonZero((long) scale - drop);
4033                 compactVal = divideAndRound(compactVal, LONG_TEN_POWERS_TABLE[drop], mc.
                    roundingMode.oldMode);
4034                 prec = longDigitLength(compactVal);
4035                 drop = prec - mcp;
4036             }
4037             return valueOf(compactVal, scale, prec);
4038         }
4039         return valueOf(compactVal, scale);
4040     }
4041
4042     /*
4043     * Returns a {@code BigDecimal} created from {@code BigInteger} value with
4044     * given scale rounded according to the MathContext settings
4045     */
4046     private static BigDecimal doRound(BigInteger intVal, int scale, MathContext mc) {
4047         int mcp = mc.precision;
4048         int prec = 0;
4049         if (mcp > 0) {
4050             long compactVal = compactValFor(intVal);
4051             int mode = mc.roundingMode.oldMode;
4052             int drop;
4053             if (compactVal == INFLATED) {
4054                 prec = bigDigitLength(intVal);
4055                 drop = prec - mcp;
4056                 while (drop > 0) {
4057                     scale = checkScaleNonZero((long) scale - drop);
4058                     intVal = divideAndRoundByTenPow(intVal, drop, mode);
4059                     compactVal = compactValFor(intVal);
4060                     if (compactVal != INFLATED) {
4061                         break;
4062                     }
4063                     prec = bigDigitLength(intVal);
4064                     drop = prec - mcp;
4065                 }
4066             }
4067             if (compactVal != INFLATED) {
4068                 prec = longDigitLength(compactVal);
4069                 drop = prec - mcp; // drop can't be more than 18
4070                 while (drop > 0) {
4071                     scale = checkScaleNonZero((long) scale - drop);
4072                     compactVal = divideAndRound(compactVal, LONG_TEN_POWERS_TABLE[drop], mc.
                        roundingMode.oldMode);
4073                     prec = longDigitLength(compactVal);
4074                     drop = prec - mcp;
4075                 }
4076                 return valueOf(compactVal, scale, prec);

```

```

4077     }
4078 }
4079     return new BigDecimal(intVal, INFLATED, scale, prec);
4080 }
4081
4082 /*
4083  * Divides {@code BigInteger} value by ten power.
4084  */
4085 private static BigInteger divideAndRoundByTenPow(BigInteger intVal, int tenPow, int
    roundingMode) {
4086     if (tenPow < LONG_TEN_POWERS_TABLE.length)
4087         intVal = divideAndRound(intVal, LONG_TEN_POWERS_TABLE[tenPow], roundingMode);
4088     else
4089         intVal = divideAndRound(intVal, bigTenToThe(tenPow), roundingMode);
4090     return intVal;
4091 }
4092
4093 /**
4094  * Internally used for division operation for division {@code long} by
4095  * {@code long}.
4096  * The returned {@code BigDecimal} object is the quotient whose scale is set
4097  * to the passed in scale. If the remainder is not zero, it will be rounded
4098  * based on the passed in roundingMode. Also, if the remainder is zero and
4099  * the last parameter, i.e. preferredScale is NOT equal to scale, the
4100  * trailing zeros of the result is stripped to match the preferredScale.
4101  */
4102 private static BigDecimal divideAndRound(long ldividend, long ldivisor, int scale, int
    roundingMode,
4103                                         int preferredScale) {
4104
4105     int qsign; // quotient sign
4106     long q = ldividend / ldivisor; // store quotient in long
4107     if (roundingMode == ROUND_DOWN && scale == preferredScale)
4108         return valueOf(q, scale);
4109     long r = ldividend % ldivisor; // store remainder in long
4110     qsign = ((ldividend < 0) == (ldivisor < 0)) ? 1 : -1;
4111     if (r != 0) {
4112         boolean increment = needIncrement(ldivisor, roundingMode, qsign, q, r);
4113         return valueOf((increment ? q + qsign : q), scale);
4114     } else {
4115         if (preferredScale != scale)
4116             return createAndStripZerosToMatchScale(q, scale, preferredScale);
4117         else
4118             return valueOf(q, scale);
4119     }
4120 }
4121
4122 /**
4123  * Divides {@code long} by {@code long} and do rounding based on the
4124  * passed in roundingMode.
4125  */
4126 private static long divideAndRound(long ldividend, long ldivisor, int roundingMode) {
4127     int qsign; // quotient sign

```

```

4128     long q = ldividend / ldivisor; // store quotient in long
4129     if (roundingMode == ROUND_DOWN)
4130         return q;
4131     long r = ldividend % ldivisor; // store remainder in long
4132     qsign = ((ldividend < 0) == (ldivisor < 0)) ? 1 : -1;
4133     if (r != 0) {
4134         boolean increment = needIncrement(ldivisor, roundingMode, qsign, q, r);
4135         return increment ? q + qsign : q;
4136     } else {
4137         return q;
4138     }
4139 }
4140
4141 /**
4142  * Shared logic of need increment computation.
4143  */
4144 private static boolean commonNeedIncrement(int roundingMode, int qsign,
4145                                           int cmpFracHalf, boolean oddQuot) {
4146     switch(roundingMode) {
4147     case ROUND_UNNECESSARY:
4148         throw new ArithmeticException("Rounding necessary");
4149
4150     case ROUND_UP: // Away from zero
4151         return true;
4152
4153     case ROUND_DOWN: // Towards zero
4154         return false;
4155
4156     case ROUND_CEILING: // Towards +infinity
4157         return qsign > 0;
4158
4159     case ROUND_FLOOR: // Towards -infinity
4160         return qsign < 0;
4161
4162     default: // Some kind of half-way rounding
4163         assert roundingMode >= ROUND_HALF_UP &&
4164             roundingMode <= ROUND_HALF_EVEN: "Unexpected rounding mode" + RoundingMode.valueOf(
4165                 roundingMode);
4166
4167         if (cmpFracHalf < 0 ) // We're closer to higher digit
4168             return false;
4169         else if (cmpFracHalf > 0 ) // We're closer to lower digit
4170             return true;
4171         else { // half-way
4172             assert cmpFracHalf == 0;
4173
4174             switch(roundingMode) {
4175             case ROUND_HALF_DOWN:
4176                 return false;
4177
4178             case ROUND_HALF_UP:
4179                 return true;

```

```

4180         case ROUND_HALF_EVEN:
4181             return oddQuot;
4182
4183         default:
4184             throw new AssertionError("Unexpected rounding mode" + roundingMode);
4185     }
4186 }
4187 }
4188 }
4189
4190 /**
4191  * Tests if quotient has to be incremented according the roundingMode
4192  */
4193 private static boolean needIncrement(long ldivisor, int roundingMode,
4194                                     int qsign, long q, long r) {
4195     assert r != 0L;
4196
4197     int cmpFracHalf;
4198     if (r <= HALF_LONG_MIN_VALUE || r > HALF_LONG_MAX_VALUE) {
4199         cmpFracHalf = 1; // 2 * r can't fit into long
4200     } else {
4201         cmpFracHalf = longCompareMagnitude(2 * r, ldivisor);
4202     }
4203
4204     return commonNeedIncrement(roundingMode, qsign, cmpFracHalf, (q & 1L) != 0L);
4205 }
4206
4207 /**
4208  * Divides {@code BigInteger} value by {@code long} value and
4209  * do rounding based on the passed in roundingMode.
4210  */
4211 private static BigInteger divideAndRound(BigInteger bdividend, long ldivisor, int roundingMode)
4212 {
4213     boolean isRemainderZero; // record remainder is zero or not
4214     int qsign; // quotient sign
4215     long r = 0; // store quotient & remainder in long
4216     MutableBigInteger mq = null; // store quotient
4217     // Descend into mutables for faster remainder checks
4218     MutableBigInteger mdividend = new MutableBigInteger(bdividend.mag);
4219     mq = new MutableBigInteger();
4220     r = mdividend.divide(ldivisor, mq);
4221     isRemainderZero = (r == 0);
4222     qsign = (ldivisor < 0) ? -bdividend.signum : bdividend.signum;
4223     if (!isRemainderZero) {
4224         if (needIncrement(ldivisor, roundingMode, qsign, mq, r)) {
4225             mq.add(MutableBigInteger.ONE);
4226         }
4227     }
4228     return mq.toBigInteger(qsign);
4229 }
4230
4231 /**
4232  * Internally used for division operation for division {@code BigInteger}

```



```

4232     * by {@code long}.
4233     * The returned {@code BigDecimal} object is the quotient whose scale is set
4234     * to the passed in scale. If the remainder is not zero, it will be rounded
4235     * based on the passed in roundingMode. Also, if the remainder is zero and
4236     * the last parameter, i.e. preferredScale is NOT equal to scale, the
4237     * trailing zeros of the result is stripped to match the preferredScale.
4238     */
4239     private static BigDecimal divideAndRound(BigInteger bdividend,
4240                                             long ldivisor, int scale, int roundingMode, int
4241                                             preferredScale) {
4242         boolean isRemainderZero; // record remainder is zero or not
4243         int qsign; // quotient sign
4244         long r = 0; // store quotient & remainder in long
4245         MutableBigInteger mq = null; // store quotient
4246         // Descend into mutables for faster remainder checks
4247         MutableBigInteger mdividend = new MutableBigInteger(bdividend.mag);
4248         mq = new MutableBigInteger();
4249         r = mdividend.divide(ldivisor, mq);
4250         isRemainderZero = (r == 0);
4251         qsign = (ldivisor < 0) ? -bdividend.signum : bdividend.signum;
4252         if (!isRemainderZero) {
4253             if (needIncrement(ldivisor, roundingMode, qsign, mq, r)) {
4254                 mq.add(MutableBigInteger.ONE);
4255             }
4256             return mq.toBigDecimal(qsign, scale);
4257         } else {
4258             if (preferredScale != scale) {
4259                 long compactVal = mq.toCompactValue(qsign);
4260                 if (compactVal != INFLATED) {
4261                     return createAndStripZerosToMatchScale(compactVal, scale, preferredScale);
4262                 }
4263                 BigInteger intVal = mq.toBigInteger(qsign);
4264                 return createAndStripZerosToMatchScale(intVal, scale, preferredScale);
4265             } else {
4266                 return mq.toBigDecimal(qsign, scale);
4267             }
4268         }
4269     }
4270     /**
4271     * Tests if quotient has to be incremented according the roundingMode
4272     */
4273     private static boolean needIncrement(long ldivisor, int roundingMode,
4274                                         int qsign, MutableBigInteger mq, long r) {
4275         assert r != 0L;
4276
4277         int cmpFracHalf;
4278         if (r <= HALF_LONG_MIN_VALUE || r > HALF_LONG_MAX_VALUE) {
4279             cmpFracHalf = 1; // 2 * r can't fit into long
4280         } else {
4281             cmpFracHalf = longCompareMagnitude(2 * r, ldivisor);
4282         }
4283     }

```

```

4284         return commonNeedIncrement(roundingMode, qsign, cmpFracHalf, mq.isOdd());
4285     }
4286
4287     /**
4288      * Divides {@code BigInteger} value by {@code BigInteger} value and
4289      * do rounding based on the passed in roundingMode.
4290      */
4291     private static BigInteger divideAndRound(BigInteger bdividend, BigInteger bdivisor, int
        roundingMode) {
4292         boolean isRemainderZero; // record remainder is zero or not
4293         int qsign; // quotient sign
4294         // Descend into mutables for faster remainder checks
4295         MutableBigInteger mdividend = new MutableBigInteger(bdividend.mag);
4296         MutableBigInteger mq = new MutableBigInteger();
4297         MutableBigInteger mdivisor = new MutableBigInteger(bdivisor.mag);
4298         MutableBigInteger mr = mdividend.divide(mdivisor, mq);
4299         isRemainderZero = mr.isZero();
4300         qsign = (bdividend.signum != bdivisor.signum) ? -1 : 1;
4301         if (!isRemainderZero) {
4302             if (needIncrement(mdivisor, roundingMode, qsign, mq, mr)) {
4303                 mq.add(MutableBigInteger.ONE);
4304             }
4305         }
4306         return mq.toBigInteger(qsign);
4307     }
4308
4309     /**
4310      * Internally used for division operation for division {@code BigInteger}
4311      * by {@code BigInteger}.
4312      * The returned {@code BigDecimal} object is the quotient whose scale is set
4313      * to the passed in scale. If the remainder is not zero, it will be rounded
4314      * based on the passed in roundingMode. Also, if the remainder is zero and
4315      * the last parameter, i.e. preferredScale is NOT equal to scale, the
4316      * trailing zeros of the result is stripped to match the preferredScale.
4317      */
4318     private static BigDecimal divideAndRound(BigInteger bdividend, BigInteger bdivisor, int scale,
        int roundingMode,
4319                                             int preferredScale) {
4320         boolean isRemainderZero; // record remainder is zero or not
4321         int qsign; // quotient sign
4322         // Descend into mutables for faster remainder checks
4323         MutableBigInteger mdividend = new MutableBigInteger(bdividend.mag);
4324         MutableBigInteger mq = new MutableBigInteger();
4325         MutableBigInteger mdivisor = new MutableBigInteger(bdivisor.mag);
4326         MutableBigInteger mr = mdividend.divide(mdivisor, mq);
4327         isRemainderZero = mr.isZero();
4328         qsign = (bdividend.signum != bdivisor.signum) ? -1 : 1;
4329         if (!isRemainderZero) {
4330             if (needIncrement(mdivisor, roundingMode, qsign, mq, mr)) {
4331                 mq.add(MutableBigInteger.ONE);
4332             }
4333             return mq.toBigDecimal(qsign, scale);
4334         } else {

```

```

4335         if (preferredScale != scale) {
4336             long compactVal = mq.toCompactValue(qsign);
4337             if (compactVal != INFLATED) {
4338                 return createAndStripZerosToMatchScale(compactVal, scale, preferredScale);
4339             }
4340             BigInteger intVal = mq.toBigInteger(qsign);
4341             return createAndStripZerosToMatchScale(intVal, scale, preferredScale);
4342         } else {
4343             return mq.toBigDecimal(qsign, scale);
4344         }
4345     }
4346 }
4347
4348 /**
4349  * Tests if quotient has to be incremented according the roundingMode
4350  */
4351 private static boolean needIncrement(MutableBigInteger mdivisor, int roundingMode,
4352                                     int qsign, MutableBigInteger mq, MutableBigInteger mr) {
4353     assert !mr.isZero();
4354     int cmpFracHalf = mr.compareHalf(mdivisor);
4355     return commonNeedIncrement(roundingMode, qsign, cmpFracHalf, mq.isOdd());
4356 }
4357
4358 /**
4359  * Remove insignificant trailing zeros from this
4360  * {@code BigInteger} value until the preferred scale is reached or no
4361  * more zeros can be removed. If the preferred scale is less than
4362  * Integer.MIN_VALUE, all the trailing zeros will be removed.
4363  *
4364  * @return new {@code BigDecimal} with a scale possibly reduced
4365  * to be closed to the preferred scale.
4366  */
4367 private static BigDecimal createAndStripZerosToMatchScale(BigInteger intVal, int scale, long
4368     preferredScale) {
4369     BigInteger qr[]; // quotient-remainder pair
4370     while (intVal.compareMagnitude(BigInteger.TEN) >= 0
4371         && scale > preferredScale) {
4372         if (intVal.testBit(0))
4373             break; // odd number cannot end in 0
4374         qr = intVal.divideAndRemainder(BigInteger.TEN);
4375         if (qr[1].signum() != 0)
4376             break; // non-0 remainder
4377         intVal = qr[0];
4378         scale = checkScale(intVal, (long) scale - 1); // could Overflow
4379     }
4380     return valueOf(intVal, scale, 0);
4381 }
4382
4383 /**
4384  * Remove insignificant trailing zeros from this
4385  * {@code long} value until the preferred scale is reached or no
4386  * more zeros can be removed. If the preferred scale is less than
4387  * Integer.MIN_VALUE, all the trailing zeros will be removed.

```

```

4387     *
4388     * @return new {@code BigDecimal} with a scale possibly reduced
4389     * to be closed to the preferred scale.
4390     */
4391     private static BigDecimal createAndStripZerosToMatchScale(long compactVal, int scale, long
        preferredScale) {
4392         while (Math.abs(compactVal) >= 10L && scale > preferredScale) {
4393             if ((compactVal & 1L) != 0L)
4394                 break; // odd number cannot end in 0
4395             long r = compactVal % 10L;
4396             if (r != 0L)
4397                 break; // non-0 remainder
4398             compactVal /= 10;
4399             scale = checkScale(compactVal, (long) scale - 1); // could Overflow
4400         }
4401         return valueOf(compactVal, scale);
4402     }
4403
4404     private static BigDecimal stripZerosToMatchScale(BigInteger intVal, long intCompact, int scale,
        int preferredScale) {
4405         if (intCompact != INFLATED) {
4406             return createAndStripZerosToMatchScale(intCompact, scale, preferredScale);
4407         } else {
4408             return createAndStripZerosToMatchScale(intVal == null ? INFLATED_BIGINT : intVal,
4409                 scale, preferredScale);
4410         }
4411     }
4412
4413     /*
4414     * returns INFLATED if overflow
4415     */
4416     private static long add(long xs, long ys){
4417         long sum = xs + ys;
4418         // See "Hacker's Delight" section 2-12 for explanation of
4419         // the overflow test.
4420         if ( (((sum ^ xs) & (sum ^ ys))) >= 0L) { // not overflowed
4421             return sum;
4422         }
4423         return INFLATED;
4424     }
4425
4426     private static BigDecimal add(long xs, long ys, int scale){
4427         long sum = add(xs, ys);
4428         if (sum != INFLATED)
4429             return BigDecimal.valueOf(sum, scale);
4430         return new BigDecimal(BigInteger.valueOf(xs).add(ys), scale);
4431     }
4432
4433     private static BigDecimal add(final long xs, int scale1, final long ys, int scale2) {
4434         long sdiff = (long) scale1 - scale2;
4435         if (sdiff == 0) {
4436             return add(xs, ys, scale1);
4437         } else if (sdiff < 0) {

```

```

4438         int raise = checkScale(xs,-sdiff);
4439         long scaledX = longMultiplyPowerTen(xs, raise);
4440         if (scaledX != INFLATED) {
4441             return add(scaledX, ys, scale2);
4442         } else {
4443             BigInteger bigsum = bigMultiplyPowerTen(xs,raise).add(ys);
4444             return ((xs^ys)>=0) ? // same sign test
4445                 new BigDecimal(bigsum, INFLATED, scale2, 0)
4446                 : valueOf(bigsum, scale2, 0);
4447         }
4448     } else {
4449         int raise = checkScale(ys,sdiff);
4450         long scaledY = longMultiplyPowerTen(ys, raise);
4451         if (scaledY != INFLATED) {
4452             return add(xs, scaledY, scale1);
4453         } else {
4454             BigInteger bigsum = bigMultiplyPowerTen(ys,raise).add(xs);
4455             return ((xs^ys)>=0) ?
4456                 new BigDecimal(bigsum, INFLATED, scale1, 0)
4457                 : valueOf(bigsum, scale1, 0);
4458         }
4459     }
4460 }

4461 private static BigDecimal add(final long xs, int scale1, BigInteger snd, int scale2) {
4462     int rscale = scale1;
4463     long sdiff = (long)rscale - scale2;
4464     boolean sameSigns = (Long.signum(xs) == snd.signum());
4465     BigInteger sum;
4466     if (sdiff < 0) {
4467         int raise = checkScale(xs,-sdiff);
4468         rscale = scale2;
4469         long scaledX = longMultiplyPowerTen(xs, raise);
4470         if (scaledX == INFLATED) {
4471             sum = snd.add(bigMultiplyPowerTen(xs,raise));
4472         } else {
4473             sum = snd.add(scaledX);
4474         }
4475     } else { //if (sdiff > 0) {
4476         int raise = checkScale(snd,sdiff);
4477         snd = bigMultiplyPowerTen(snd,raise);
4478         sum = snd.add(xs);
4479     }
4480     return (sameSigns) ?
4481         new BigDecimal(sum, INFLATED, rscale, 0) :
4482         valueOf(sum, rscale, 0);
4483 }

4484
4485 private static BigDecimal add(BigInteger fst, int scale1, BigInteger snd, int scale2) {
4486     int rscale = scale1;
4487     long sdiff = (long)rscale - scale2;
4488     if (sdiff != 0) {
4489         if (sdiff < 0) {

```

```

4491         int raise = checkScale(fst,-sdiff);
4492         rscale = scale2;
4493         fst = bigMultiplyPowerTen(fst,raise);
4494     } else {
4495         int raise = checkScale(snd,sdiff);
4496         snd = bigMultiplyPowerTen(snd,raise);
4497     }
4498 }
4499 BigInteger sum = fst.add(snd);
4500 return (fst.signum == snd.signum) ?
4501     new BigDecimal(sum, INFLATED, rscale, 0) :
4502     valueOf(sum, rscale, 0);
4503 }
4504
4505 private static BigInteger bigMultiplyPowerTen(long value, int n) {
4506     if (n <= 0)
4507         return BigInteger.valueOf(value);
4508     return bigTenToThe(n).multiply(value);
4509 }
4510
4511 private static BigInteger bigMultiplyPowerTen(BigInteger value, int n) {
4512     if (n <= 0)
4513         return value;
4514     if (n < LONG_TEN_POWERS_TABLE.length) {
4515         return value.multiply(LONG_TEN_POWERS_TABLE[n]);
4516     }
4517     return value.multiply(bigTenToThe(n));
4518 }
4519
4520 /**
4521  * Returns a {@code BigDecimal} whose value is {@code (xs /
4522  * ys)}, with rounding according to the context settings.
4523  *
4524  * Fast path - used only when (xscale <= yscale && yscale < 18
4525  * && mc.precision<18) {
4526  */
4527 private static BigDecimal divideSmallFastPath(final long xs, int xscale,
4528         final long ys, int yscale,
4529         long preferredScale, MathContext mc) {
4530     int mcp = mc.precision;
4531     int roundingMode = mc.roundingMode.oldMode;
4532
4533     assert (xscale <= yscale) && (yscale < 18) && (mcp < 18);
4534     int xraise = yscale - xscale; // xraise >=0
4535     long scaledX = (xraise==0) ? xs :
4536         longMultiplyPowerTen(xs, xraise); // can't overflow here!
4537     BigDecimal quotient;
4538
4539     int cmp = longCompareMagnitude(scaledX, ys);
4540     if (cmp > 0) { // satisfy constraint (b)
4541         yscale -= 1; // [that is, divisor *= 10]
4542         int scl = checkScaleNonZero(preferredScale + yscale - xscale + mcp);
4543         if (checkScaleNonZero((long) mcp + yscale - xscale) > 0) {

```

```

4544         // assert newScale >= xscale
4545         int raise = checkScaleNonZero((long) mcp + yscale - xscale);
4546         long scaledXs;
4547         if ((scaledXs = longMultiplyPowerTen(xs, raise)) == INFLATED) {
4548             quotient = null;
4549             if ((mcp-1) >= 0 && (mcp-1) < LONG_TEN_POWERS_TABLE.length) {
4550                 quotient = multiplyDivideAndRound(LONG_TEN_POWERS_TABLE[mcp-1], scaledX, ys
4551                     , scl, roundingMode, checkScaleNonZero(preferredScale));
4552             }
4553             if (quotient == null) {
4554                 BigInteger rb = bigMultiplyPowerTen(scaledX, mcp-1);
4555                 quotient = divideAndRound(rb, ys,
4556                     scl, roundingMode, checkScaleNonZero(
4557                         preferredScale));
4558             } else {
4559                 quotient = divideAndRound(scaledXs, ys, scl, roundingMode, checkScaleNonZero(
4560                     preferredScale));
4561             }
4562         } else {
4563             int newScale = checkScaleNonZero((long) xscale - mcp);
4564             // assert newScale >= yscale
4565             if (newScale == yscale) { // easy case
4566                 quotient = divideAndRound(xs, ys, scl, roundingMode, checkScaleNonZero(
4567                     preferredScale));
4568             } else {
4569                 int raise = checkScaleNonZero((long) newScale - yscale);
4570                 long scaledYs;
4571                 if ((scaledYs = longMultiplyPowerTen(ys, raise)) == INFLATED) {
4572                     BigInteger rb = bigMultiplyPowerTen(ys, raise);
4573                     quotient = divideAndRound(BigInteger.valueOf(xs),
4574                         rb, scl, roundingMode, checkScaleNonZero(
4575                             preferredScale));
4576                 } else {
4577                     quotient = divideAndRound(xs, scaledYs, scl, roundingMode, checkScaleNonZero(
4578                         preferredScale));
4579                 }
4580             }
4581         }
4582     } else {
4583         // abs(scaledX) <= abs(ys)
4584         // result is "scaledX * 10^msp / ys"
4585         int scl = checkScaleNonZero(preferredScale + yscale - xscale + mcp);
4586         if (cmp == 0) {
4587             // abs(scaleX) == abs(ys) => result will be scaled 10^mcp + correct sign
4588             quotient = roundedTenPower(((scaledX < 0) == (ys < 0)) ? 1 : -1, mcp, scl,
4589                 checkScaleNonZero(preferredScale));
4590         } else {
4591             // abs(scaledX) < abs(ys)
4592             long scaledXs;
4593             if ((scaledXs = longMultiplyPowerTen(scaledX, mcp)) == INFLATED) {
4594                 quotient = null;
4595                 if (mcp < LONG_TEN_POWERS_TABLE.length) {

```

```

4590         quotient = multiplyDivideAndRound(LONG_TEN_POWERS_TABLE[mcp], scaledX, ys,
4591                                           scl, roundingMode, checkScaleNonZero(preferredScale));
4592     }
4593     if(quotient==null) {
4594         BigInteger rb = bigMultiplyPowerTen(scaledX,mcp);
4595         quotient = divideAndRound(rb, ys,
4596                                   scl, roundingMode, checkScaleNonZero(
4597                                       preferredScale));
4598     }
4599     } else {
4600         quotient = divideAndRound(scaledXs, ys, scl, roundingMode, checkScaleNonZero(
4601             preferredScale));
4602     }
4603 }
4604 // doRound, here, only affects 1000000000 case.
4605 return doRound(quotient,mc);
4606 }
4607 /**
4608  * Returns a {@code BigDecimal} whose value is {@code (xs /
4609  * ys)}, with rounding according to the context settings.
4610  */
4611 private static BigDecimal divide(final long xs, int xscale, final long ys, int yscale, long
4612     preferredScale, MathContext mc) {
4613     int mcp = mc.precision;
4614     if(xscale <= yscale && yscale < 18 && mcp<18) {
4615         return divideSmallFastPath(xs, xscale, ys, yscale, preferredScale, mc);
4616     }
4617     if (compareMagnitudeNormalized(xs, xscale, ys, yscale) > 0) { // satisfy constraint (b)
4618         yscale -= 1; // [that is, divisor *= 10]
4619     }
4620     int roundingMode = mc.roundingMode.oldMode;
4621     // In order to find out whether the divide generates the exact result,
4622     // we avoid calling the above divide method. 'quotient' holds the
4623     // return BigDecimal object whose scale will be set to 'scl'.
4624     int scl = checkScaleNonZero(preferredScale + yscale - xscale + mcp);
4625     BigDecimal quotient;
4626     if (checkScaleNonZero((long) mcp + yscale - xscale) > 0) {
4627         int raise = checkScaleNonZero((long) mcp + yscale - xscale);
4628         long scaledXs;
4629         if ((scaledXs = longMultiplyPowerTen(xs, raise)) == INFLATED) {
4630             BigInteger rb = bigMultiplyPowerTen(xs,raise);
4631             quotient = divideAndRound(rb, ys, scl, roundingMode, checkScaleNonZero(
4632                 preferredScale));
4633         } else {
4634             quotient = divideAndRound(scaledXs, ys, scl, roundingMode, checkScaleNonZero(
4635                 preferredScale));
4636         }
4637     } else {
4638         int newScale = checkScaleNonZero((long) xscale - mcp);
4639         // assert newScale >= yscale
4640         if (newScale == yscale) { // easy case

```



```

4637         quotient = divideAndRound(xs, ys, scl, roundingMode, checkScaleNonZero(
4638             preferredScale));
4639     } else {
4640         int raise = checkScaleNonZero((long) newScale - yscale);
4641         long scaledYs;
4642         if ((scaledYs = longMultiplyPowerTen(ys, raise)) == INFLATED) {
4643             BigInteger rb = bigMultiplyPowerTen(ys, raise);
4644             quotient = divideAndRound(BigInteger.valueOf(xs),
4645                                     rb, scl, roundingMode, checkScaleNonZero(
4646                                         preferredScale));
4647         } else {
4648             quotient = divideAndRound(xs, scaledYs, scl, roundingMode, checkScaleNonZero(
4649                 preferredScale));
4650         }
4651     }
4652     // doRound, here, only affects 1000000000 case.
4653     return doRound(quotient, mc);
4654 }
4655 /**
4656  * Returns a {@code BigDecimal} whose value is {@code (xs /
4657  * ys)}, with rounding according to the context settings.
4658  */
4659 private static BigDecimal divide(BigInteger xs, int xscale, long ys, int yscale, long
4660     preferredScale, MathContext mc) {
4661     // Normalize dividend & divisor so that both fall into [0.1, 0.999...]
4662     if ((-compareMagnitudeNormalized(ys, yscale, xs, xscale)) > 0) { // satisfy constraint (b)
4663         yscale -= 1; // [that is, divisor *= 10]
4664     }
4665     int mcp = mc.precision;
4666     int roundingMode = mc.roundingMode.oldMode;
4667
4668     // In order to find out whether the divide generates the exact result,
4669     // we avoid calling the above divide method. 'quotient' holds the
4670     // return BigDecimal object whose scale will be set to 'scl'.
4671     BigDecimal quotient;
4672     int scl = checkScaleNonZero(preferredScale + yscale - xscale + mcp);
4673     if (checkScaleNonZero((long) mcp + yscale - xscale) > 0) {
4674         int raise = checkScaleNonZero((long) mcp + yscale - xscale);
4675         BigInteger rb = bigMultiplyPowerTen(xs, raise);
4676         quotient = divideAndRound(rb, ys, scl, roundingMode, checkScaleNonZero(preferredScale))
4677             ;
4678     } else {
4679         int newScale = checkScaleNonZero((long) xscale - mcp);
4680         // assert newScale >= yscale
4681         if (newScale == yscale) { // easy case
4682             quotient = divideAndRound(xs, ys, scl, roundingMode, checkScaleNonZero(
4683                 preferredScale));
4684         } else {
4685             int raise = checkScaleNonZero((long) newScale - yscale);
4686             long scaledYs;
4687             if ((scaledYs = longMultiplyPowerTen(ys, raise)) == INFLATED) {

```

```

4684         BigInteger rb = bigMultiplyPowerTen(ys,raise);
4685         quotient = divideAndRound(xs, rb, scl, roundingMode,checkScaleNonZero(
            preferredScale));
4686     } else {
4687         quotient = divideAndRound(xs, scaledYs, scl, roundingMode,checkScaleNonZero(
            preferredScale));
4688     }
4689 }
4690 }
4691 // doRound, here, only affects 1000000000 case.
4692 return doRound(quotient, mc);
4693 }
4694
4695 /**
4696  * Returns a {@code BigDecimal} whose value is {@code (xs /
4697  * ys)}, with rounding according to the context settings.
4698  */
4699 private static BigDecimal divide(long xs, int xscale, BigInteger ys, int yscale, long
    preferredScale, MathContext mc) {
4700     // Normalize dividend & divisor so that both fall into [0.1, 0.999...]
4701     if (compareMagnitudeNormalized(xs, xscale, ys, yscale) > 0) { // satisfy constraint (b)
4702         yscale -= 1; // [that is, divisor *= 10]
4703     }
4704     int mcp = mc.precision;
4705     int roundingMode = mc.roundingMode.oldMode;
4706
4707     // In order to find out whether the divide generates the exact result,
4708     // we avoid calling the above divide method. 'quotient' holds the
4709     // return BigDecimal object whose scale will be set to 'scl'.
4710     BigDecimal quotient;
4711     int scl = checkScaleNonZero(preferredScale + yscale - xscale + mcp);
4712     if (checkScaleNonZero((long) mcp + yscale - xscale) > 0) {
4713         int raise = checkScaleNonZero((long) mcp + yscale - xscale);
4714         BigInteger rb = bigMultiplyPowerTen(xs,raise);
4715         quotient = divideAndRound(rb, ys, scl, roundingMode, checkScaleNonZero(preferredScale))
            ;
4716     } else {
4717         int newScale = checkScaleNonZero((long) xscale - mcp);
4718         int raise = checkScaleNonZero((long) newScale - yscale);
4719         BigInteger rb = bigMultiplyPowerTen(ys,raise);
4720         quotient = divideAndRound(BigInteger.valueOf(xs), rb, scl, roundingMode,
            checkScaleNonZero(preferredScale));
4721     }
4722     // doRound, here, only affects 1000000000 case.
4723     return doRound(quotient, mc);
4724 }
4725
4726 /**
4727  * Returns a {@code BigDecimal} whose value is {@code (xs /
4728  * ys)}, with rounding according to the context settings.
4729  */
4730 private static BigDecimal divide(BigInteger xs, int xscale, BigInteger ys, int yscale, long
    preferredScale, MathContext mc) {

```

```

4731     // Normalize dividend & divisor so that both fall into [0.1, 0.999...]
4732     if (compareMagnitudeNormalized(xs, xscale, ys, yscale) > 0) { // satisfy constraint (b)
4733         yscale -= 1; // [that is, divisor *= 10]
4734     }
4735     int mcp = mc.precision;
4736     int roundingMode = mc.roundingMode.oldMode;
4737
4738     // In order to find out whether the divide generates the exact result,
4739     // we avoid calling the above divide method. 'quotient' holds the
4740     // return BigDecimal object whose scale will be set to 'scl'.
4741     BigDecimal quotient;
4742     int scl = checkScaleNonZero(preferredScale + yscale - xscale + mcp);
4743     if (checkScaleNonZero((long) mcp + yscale - xscale) > 0) {
4744         int raise = checkScaleNonZero((long) mcp + yscale - xscale);
4745         BigInteger rb = bigMultiplyPowerTen(xs, raise);
4746         quotient = divideAndRound(rb, ys, scl, roundingMode, checkScaleNonZero(preferredScale));
4747     } else {
4748         int newScale = checkScaleNonZero((long) xscale - mcp);
4749         int raise = checkScaleNonZero((long) newScale - yscale);
4750         BigInteger rb = bigMultiplyPowerTen(ys, raise);
4751         quotient = divideAndRound(xs, rb, scl, roundingMode, checkScaleNonZero(preferredScale));
4752     }
4753     // doRound, here, only affects 1000000000 case.
4754     return doRound(quotient, mc);
4755 }
4756
4757 /*
4758  * performs divideAndRound for (dividend0*dividend1, divisor)
4759  * returns null if quotient can't fit into long value;
4760  */
4761 private static BigDecimal multiplyDivideAndRound(long dividend0, long dividend1, long divisor,
4762         int scale, int roundingMode,
4763         int preferredScale) {
4764     int qsign = Long.signum(dividend0)*Long.signum(dividend1)*Long.signum(divisor);
4765     dividend0 = Math.abs(dividend0);
4766     dividend1 = Math.abs(dividend1);
4767     divisor = Math.abs(divisor);
4768     // multiply dividend0 * dividend1
4769     long d0_hi = dividend0 >>> 32;
4770     long d0_lo = dividend0 & LONG_MASK;
4771     long d1_hi = dividend1 >>> 32;
4772     long d1_lo = dividend1 & LONG_MASK;
4773     long product = d0_lo * d1_lo;
4774     long d0 = product & LONG_MASK;
4775     long d1 = product >>> 32;
4776     product = d0_hi * d1_lo + d1;
4777     d1 = product & LONG_MASK;
4778     long d2 = product >>> 32;
4779     product = d0_lo * d1_hi + d1;
4780     d1 = product & LONG_MASK;
4781     d2 += product >>> 32;
4782     long d3 = d2>>>32;

```

```

4782         d2 &= LONG_MASK;
4783         product = d0_hi*d1_hi + d2;
4784         d2 = product & LONG_MASK;
4785         d3 = ((product>>>32) + d3) & LONG_MASK;
4786         final long dividendHi = make64(d3,d2);
4787         final long dividendLo = make64(d1,d0);
4788         // divide
4789         return divideAndRound128(dividendHi, dividendLo, divisor, qsign, scale, roundingMode,
            preferredScale);
4790     }
4791
4792     private static final long DIV_NUM_BASE = (1L<<32); // Number base (32 bits).
4793
4794     /*
4795     * divideAndRound 128-bit value by long divisor.
4796     * returns null if quotient can't fit into long value;
4797     * Specialized version of Knuth's division
4798     */
4799     private static BigDecimal divideAndRound128(final long dividendHi, final long dividendLo, long
        divisor, int sign,
4800                                                int scale, int roundingMode, int preferredScale) {
4801         if (dividendHi >= divisor) {
4802             return null;
4803         }
4804
4805         final int shift = Long.numberOfLeadingZeros(divisor);
4806         divisor <<= shift;
4807
4808         final long v1 = divisor >>> 32;
4809         final long v0 = divisor & LONG_MASK;
4810
4811         long tmp = dividendLo << shift;
4812         long u1 = tmp >>> 32;
4813         long u0 = tmp & LONG_MASK;
4814
4815         tmp = (dividendHi << shift) | (dividendLo >>> 64 - shift);
4816         long u2 = tmp & LONG_MASK;
4817         long q1, r_tmp;
4818         if (v1 == 1) {
4819             q1 = tmp;
4820             r_tmp = 0;
4821         } else if (tmp >= 0) {
4822             q1 = tmp / v1;
4823             r_tmp = tmp - q1 * v1;
4824         } else {
4825             long[] rq = divRemNegativeLong(tmp, v1);
4826             q1 = rq[1];
4827             r_tmp = rq[0];
4828         }
4829
4830         while(q1 >= DIV_NUM_BASE || unsignedLongCompare(q1*v0, make64(r_tmp, u1))) {
4831             q1--;
4832             r_tmp += v1;

```

```

4833         if (r_tmp >= DIV_NUM_BASE)
4834             break;
4835     }
4836
4837     tmp = mulsub(u2,u1,v1,v0,q1);
4838     u1 = tmp & LONG_MASK;
4839     long q0;
4840     if (v1 == 1) {
4841         q0 = tmp;
4842         r_tmp = 0;
4843     } else if (tmp >= 0) {
4844         q0 = tmp / v1;
4845         r_tmp = tmp - q0 * v1;
4846     } else {
4847         long[] rq = divRemNegativeLong(tmp, v1);
4848         q0 = rq[1];
4849         r_tmp = rq[0];
4850     }
4851
4852     while(q0 >= DIV_NUM_BASE || unsignedLongCompare(q0*v0,make64(r_tmp,u0))) {
4853         q0--;
4854         r_tmp += v1;
4855         if (r_tmp >= DIV_NUM_BASE)
4856             break;
4857     }
4858
4859     if((int)q1 < 0) {
4860         // result (which is positive and unsigned here)
4861         // can't fit into long due to sign bit is used for value
4862         MutableBigInteger mq = new MutableBigInteger(new int[]{(int)q1, (int)q0});
4863         if (roundingMode == ROUND_DOWN && scale == preferredScale) {
4864             return mq.toBigDecimal(sign, scale);
4865         }
4866         long r = mulsub(u1, u0, v1, v0, q0) >>> shift;
4867         if (r != 0) {
4868             if(needIncrement(divisor >>> shift, roundingMode, sign, mq, r)){
4869                 mq.add(MutableBigInteger.ONE);
4870             }
4871             return mq.toBigDecimal(sign, scale);
4872         } else {
4873             if (preferredScale != scale) {
4874                 BigInteger intVal = mq.toBigInteger(sign);
4875                 return createAndStripZerosToMatchScale(intVal,scale, preferredScale);
4876             } else {
4877                 return mq.toBigDecimal(sign, scale);
4878             }
4879         }
4880     }
4881
4882     long q = make64(q1,q0);
4883     q*=sign;
4884
4885     if (roundingMode == ROUND_DOWN && scale == preferredScale)

```

```

4886         return valueOf(q, scale);
4887
4888         long r = mulsub(u1, u0, v1, v0, q0) >>> shift;
4889         if (r != 0) {
4890             boolean increment = needIncrement(divisor >>> shift, roundingMode, sign, q, r);
4891             return valueOf((increment ? q + sign : q), scale);
4892         } else {
4893             if (preferredScale != scale) {
4894                 return createAndStripZerosToMatchScale(q, scale, preferredScale);
4895             } else {
4896                 return valueOf(q, scale);
4897             }
4898         }
4899     }
4900
4901     /*
4902     * calculate divideAndRound for ldividend*10raise / divisor
4903     * when abs(dividend)==abs(divisor);
4904     */
4905     private static BigDecimal roundedTenPower(int qsign, int raise, int scale, int preferredScale)
4906     {
4907         if (scale > preferredScale) {
4908             int diff = scale - preferredScale;
4909             if (diff < raise) {
4910                 return scaledTenPow(raise - diff, qsign, preferredScale);
4911             } else {
4912                 return valueOf(qsign, scale - raise);
4913             }
4914         } else {
4915             return scaledTenPow(raise, qsign, scale);
4916         }
4917     }
4918
4919     static BigDecimal scaledTenPow(int n, int sign, int scale) {
4920         if (n < LONG_TEN_POWERS_TABLE.length)
4921             return valueOf(sign * LONG_TEN_POWERS_TABLE[n], scale);
4922         else {
4923             BigInteger unscaledVal = bigTenToThe(n);
4924             if (sign == -1) {
4925                 unscaledVal = unscaledVal.negate();
4926             }
4927             return new BigDecimal(unscaledVal, INFLATED, scale, n + 1);
4928         }
4929     }
4930
4931     /**
4932     * Calculate the quotient and remainder of dividing a negative long by
4933     * another long.
4934     *
4935     * @param n the numerator; must be negative
4936     * @param d the denominator; must not be unity
4937     * @return a two-element {@code long} array with the remainder and quotient in
4938     *         the initial and final elements, respectively

```

```

4938     */
4939     private static long[] divRemNegativeLong(long n, long d) {
4940         assert n < 0 : "Non-negative numerator " + n;
4941         assert d != 1 : "Unity denominator";
4942
4943         // Approximate the quotient and remainder
4944         long q = (n >>> 1) / (d >>> 1);
4945         long r = n - q * d;
4946
4947         // Correct the approximation
4948         while (r < 0) {
4949             r += d;
4950             q--;
4951         }
4952         while (r >= d) {
4953             r -= d;
4954             q++;
4955         }
4956
4957         // n - q*d == r && 0 <= r < d, hence we're done.
4958         return new long[] {r, q};
4959     }
4960
4961     private static long make64(long hi, long lo) {
4962         return hi<<32 | lo;
4963     }
4964
4965     private static long mulsub(long u1, long u0, final long v1, final long v0, long q0) {
4966         long tmp = u0 - q0*v0;
4967         return make64(u1 + (tmp>>>32) - q0*v1, tmp & LONG_MASK);
4968     }
4969
4970     private static boolean unsignedLongCompare(long one, long two) {
4971         return (one+Long.MIN_VALUE) > (two+Long.MIN_VALUE);
4972     }
4973
4974     private static boolean unsignedLongCompareEq(long one, long two) {
4975         return (one+Long.MIN_VALUE) >= (two+Long.MIN_VALUE);
4976     }
4977
4978     // Compare Normalize dividend & divisor so that both fall into [0.1, 0.999...]
4979     private static int compareMagnitudeNormalized(long xs, int xscale, long ys, int yscale) {
4980         // assert xs!=0 && ys!=0
4981         int sdiff = xscale - yscale;
4982         if (sdiff != 0) {
4983             if (sdiff < 0) {
4984                 xs = longMultiplyPowerTen(xs, -sdiff);
4985             } else { // sdiff > 0
4986                 ys = longMultiplyPowerTen(ys, sdiff);
4987             }
4988         }
4989     }
4990     if (xs != INFLATED)

```

```

4991         return (ys != INFLATED) ? longCompareMagnitude(xs, ys) : -1;
4992     else
4993         return 1;
4994 }
4995
4996 // Compare Normalize dividend & divisor so that both fall into [0.1, 0.999...]
4997 private static int compareMagnitudeNormalized(long xs, int xscale, BigInteger ys, int yscale) {
4998     // assert "ys can't be represented as long"
4999     if (xs == 0)
5000         return -1;
5001     int sdiff = xscale - yscale;
5002     if (sdiff < 0) {
5003         if (longMultiplyPowerTen(xs, -sdiff) == INFLATED) {
5004             return bigMultiplyPowerTen(xs, -sdiff).compareMagnitude(ys);
5005         }
5006     }
5007     return -1;
5008 }
5009
5010 // Compare Normalize dividend & divisor so that both fall into [0.1, 0.999...]
5011 private static int compareMagnitudeNormalized(BigInteger xs, int xscale, BigInteger ys, int
    yscale) {
5012     int sdiff = xscale - yscale;
5013     if (sdiff < 0) {
5014         return bigMultiplyPowerTen(xs, -sdiff).compareMagnitude(ys);
5015     } else { // sdiff >= 0
5016         return xs.compareMagnitude(bigMultiplyPowerTen(ys, sdiff));
5017     }
5018 }
5019
5020 private static long multiply(long x, long y){
5021     long product = x * y;
5022     long ax = Math.abs(x);
5023     long ay = Math.abs(y);
5024     if (((ax | ay) >>> 31 == 0) || (y == 0) || (product / y == x)){
5025         return product;
5026     }
5027     return INFLATED;
5028 }
5029
5030 private static BigDecimal multiply(long x, long y, int scale) {
5031     long product = multiply(x, y);
5032     if (product != INFLATED) {
5033         return valueOf(product, scale);
5034     }
5035     return new BigDecimal(BigInteger.valueOf(x).multiply(y), INFLATED, scale, 0);
5036 }
5037
5038 private static BigDecimal multiply(long x, BigInteger y, int scale) {
5039     if (x == 0) {
5040         return zeroValueOf(scale);
5041     }
5042     return new BigDecimal(y.multiply(x), INFLATED, scale, 0);

```



```

5043     }
5044
5045     private static BigDecimal multiply(BigInteger x, BigInteger y, int scale) {
5046         return new BigDecimal(x.multiply(y), INFLATED, scale, 0);
5047     }
5048
5049     /**
5050      * Multiplies two long values and rounds according {@code MathContext}
5051      */
5052     private static BigDecimal multiplyAndRound(long x, long y, int scale, MathContext mc) {
5053         long product = multiply(x, y);
5054         if (product != INFLATED) {
5055             return doRound(product, scale, mc);
5056         }
5057         // attempt to do it in 128 bits
5058         int rsign = 1;
5059         if (x < 0) {
5060             x = -x;
5061             rsign = -1;
5062         }
5063         if (y < 0) {
5064             y = -y;
5065             rsign *= -1;
5066         }
5067         // multiply dividend0 * dividend1
5068         long m0_hi = x >>> 32;
5069         long m0_lo = x & LONG_MASK;
5070         long m1_hi = y >>> 32;
5071         long m1_lo = y & LONG_MASK;
5072         product = m0_lo * m1_lo;
5073         long m0 = product & LONG_MASK;
5074         long m1 = product >>> 32;
5075         product = m0_hi * m1_lo + m1;
5076         m1 = product & LONG_MASK;
5077         long m2 = product >>> 32;
5078         product = m0_lo * m1_hi + m1;
5079         m1 = product & LONG_MASK;
5080         m2 += product >>> 32;
5081         long m3 = m2 >>> 32;
5082         m2 &= LONG_MASK;
5083         product = m0_hi * m1_hi + m2;
5084         m2 = product & LONG_MASK;
5085         m3 = ((product >>> 32) + m3) & LONG_MASK;
5086         final long mHi = make64(m3, m2);
5087         final long mLo = make64(m1, m0);
5088         BigDecimal res = doRound128(mHi, mLo, rsign, scale, mc);
5089         if (res != null) {
5090             return res;
5091         }
5092         res = new BigDecimal(BigInteger.valueOf(x).multiply(y * rsign), INFLATED, scale, 0);
5093         return doRound(res, mc);
5094     }
5095

```

```

5096 private static BigDecimal multiplyAndRound(long x, BigInteger y, int scale, MathContext mc) {
5097     if(x==0) {
5098         return zeroValueOf(scale);
5099     }
5100     return doRound(y.multiply(x), scale, mc);
5101 }
5102
5103 private static BigDecimal multiplyAndRound(BigInteger x, BigInteger y, int scale, MathContext
5104     mc) {
5105     return doRound(x.multiply(y), scale, mc);
5106 }
5107 /**
5108  * rounds 128-bit value according {@code MathContext}
5109  * returns null if result can't be represented as compact BigDecimal.
5110  */
5111 private static BigDecimal doRound128(long hi, long lo, int sign, int scale, MathContext mc) {
5112     int mcp = mc.precision;
5113     int drop;
5114     BigDecimal res = null;
5115     if(((drop = precision(hi, lo) - mcp) > 0)&&(drop<LONG_TEN_POWERS_TABLE.length)) {
5116         scale = checkScaleNonZero((long)scale - drop);
5117         res = divideAndRound128(hi, lo, LONG_TEN_POWERS_TABLE[drop], sign, scale, mc.
5118             roundingMode.oldMode, scale);
5119     }
5120     if(res!=null) {
5121         return doRound(res,mc);
5122     }
5123     return null;
5124 }
5125
5126 private static final long[][] LONGLONG_TEN_POWERS_TABLE = {
5127     { 0L, 0x8AC7230489E80000L }, //10^19
5128     { 0x5L, 0x6bc75e2d63100000L }, //10^20
5129     { 0x36L, 0x35c9adc5dea00000L }, //10^21
5130     { 0x21eL, 0x19e0c9bab2400000L }, //10^22
5131     { 0x152dL, 0x02c7e14af6800000L }, //10^23
5132     { 0xd3c2L, 0x1bcecceda1000000L }, //10^24
5133     { 0x84595L, 0x161401484a000000L }, //10^25
5134     { 0x52b7d2L, 0xdcc80cd2e4000000L }, //10^26
5135     { 0x33b2e3cL, 0x9fd0803ce8000000L }, //10^27
5136     { 0x204fce5eL, 0x3e25026110000000L }, //10^28
5137     { 0x1431e0faeL, 0x6d7217caa0000000L }, //10^29
5138     { 0xc9f2c9cd0L, 0x4674edea40000000L }, //10^30
5139     { 0x7e37be2022L, 0xc0914b2680000000L }, //10^31
5140     { 0x4ee2d6d415bL, 0x85acef8100000000L }, //10^32
5141     { 0x314dc6448d93L, 0x38c15b0a00000000L }, //10^33
5142     { 0x1ed09bead87c0L, 0x378d8e6400000000L }, //10^34
5143     { 0x13426172c74d82L, 0x2b878fe800000000L }, //10^35
5144     { 0xc097ce7bc90715L, 0xb34b9f1000000000L }, //10^36
5145     { 0x785ee10d5da46d9L, 0x00f436a000000000L }, //10^37
5146     { 0x4b3b4ca85a86c47aL, 0x098a224000000000L }, //10^38
5147 };

```

```

5147
5148  /*
5149   * returns precision of 128-bit value
5150   */
5151  private static int precision(long hi, long lo){
5152      if(hi==0) {
5153          if(lo>=0) {
5154              return longDigitLength(lo);
5155          }
5156          return (unsignedLongCompareEq(lo, LONGLONG_TEN_POWERS_TABLE[0][1])) ? 20 : 19;
5157          // 0x8AC7230489E80000L = unsigned 2^19
5158      }
5159      int r = ((128 - Long.numberOfLeadingZeros(hi) + 1) * 1233) >>> 12;
5160      int idx = r-19;
5161      return (idx >= LONGLONG_TEN_POWERS_TABLE.length || longLongCompareMagnitude(hi, lo,
5162                                                                                     LONGLONG_TEN_POWERS_TABLE
5163                                                                                     [idx][0],
5164                                                                                     LONGLONG_TEN_POWERS_TABLE
5165                                                                                     [idx][1]))
5166          ? r : r +
5167          1;
5168  }
5169
5170  /*
5171   * returns true if 128 bit number <hi0,lo0> is less then <hi1,lo1>
5172   * hi0 & hi1 should be non-negative
5173   */
5174  private static boolean longLongCompareMagnitude(long hi0, long lo0, long hi1, long lo1) {
5175      if(hi0!=hi1) {
5176          return hi0<hi1;
5177      }
5178      return (lo0+Long.MIN_VALUE) < (lo1+Long.MIN_VALUE);
5179  }
5180
5181  private static BigDecimal divide(long dividend, int dividendScale, long divisor, int
5182      divisorScale, int scale, int roundingMode) {
5183      if (checkScale(dividend,(long)scale + divisorScale) > dividendScale) {
5184          int newScale = scale + divisorScale;
5185          int raise = newScale - dividendScale;
5186          if(raise<LONG_TEN_POWERS_TABLE.length) {
5187              long xs = dividend;
5188              if ((xs = longMultiplyPowerTen(xs, raise)) != INFLATED) {
5189                  return divideAndRound(xs, divisor, scale, roundingMode, scale);
5190              }
5191              BigDecimal q = multiplyDivideAndRound(LONG_TEN_POWERS_TABLE[raise], dividend,
5192                  divisor, scale, roundingMode, scale);
5193              if(q!=null) {
5194                  return q;
5195              }
5196          }
5197          BigInteger scaledDividend = bigMultiplyPowerTen(dividend, raise);
5198          return divideAndRound(scaledDividend, divisor, scale, roundingMode, scale);
5199      } else {

```

```

5193         int newScale = checkScale(divisor, (long)dividendScale - scale);
5194         int raise = newScale - divisorScale;
5195         if(raise < LONG_TEN_POWERS_TABLE.length) {
5196             long ys = divisor;
5197             if ((ys = longMultiplyPowerTen(ys, raise)) != INFLATED) {
5198                 return divideAndRound(dividend, ys, scale, roundingMode, scale);
5199             }
5200         }
5201         BigInteger scaledDivisor = bigMultiplyPowerTen(divisor, raise);
5202         return divideAndRound(BigInteger.valueOf(dividend), scaledDivisor, scale, roundingMode,
                    scale);
5203     }
5204 }
5205
5206 private static BigDecimal divide(BigInteger dividend, int dividendScale, long divisor, int
    divisorScale, int scale, int roundingMode) {
5207     if (checkScale(dividend, (long)scale + divisorScale) > dividendScale) {
5208         int newScale = scale + divisorScale;
5209         int raise = newScale - dividendScale;
5210         BigInteger scaledDividend = bigMultiplyPowerTen(dividend, raise);
5211         return divideAndRound(scaledDividend, divisor, scale, roundingMode, scale);
5212     } else {
5213         int newScale = checkScale(divisor, (long)dividendScale - scale);
5214         int raise = newScale - divisorScale;
5215         if(raise < LONG_TEN_POWERS_TABLE.length) {
5216             long ys = divisor;
5217             if ((ys = longMultiplyPowerTen(ys, raise)) != INFLATED) {
5218                 return divideAndRound(dividend, ys, scale, roundingMode, scale);
5219             }
5220         }
5221         BigInteger scaledDivisor = bigMultiplyPowerTen(divisor, raise);
5222         return divideAndRound(dividend, scaledDivisor, scale, roundingMode, scale);
5223     }
5224 }
5225
5226 private static BigDecimal divide(long dividend, int dividendScale, BigInteger divisor, int
    divisorScale, int scale, int roundingMode) {
5227     if (checkScale(dividend, (long)scale + divisorScale) > dividendScale) {
5228         int newScale = scale + divisorScale;
5229         int raise = newScale - dividendScale;
5230         BigInteger scaledDividend = bigMultiplyPowerTen(dividend, raise);
5231         return divideAndRound(scaledDividend, divisor, scale, roundingMode, scale);
5232     } else {
5233         int newScale = checkScale(divisor, (long)dividendScale - scale);
5234         int raise = newScale - divisorScale;
5235         BigInteger scaledDivisor = bigMultiplyPowerTen(divisor, raise);
5236         return divideAndRound(BigInteger.valueOf(dividend), scaledDivisor, scale, roundingMode,
                    scale);
5237     }
5238 }
5239
5240 private static BigDecimal divide(BigInteger dividend, int dividendScale, BigInteger divisor,
    int divisorScale, int scale, int roundingMode) {

```

```

5241     if (checkScale(dividend, (long)scale + divisorScale) > dividendScale) {
5242         int newScale = scale + divisorScale;
5243         int raise = newScale - dividendScale;
5244         BigInteger scaledDividend = bigMultiplyPowerTen(dividend, raise);
5245         return divideAndRound(scaledDividend, divisor, scale, roundingMode, scale);
5246     } else {
5247         int newScale = checkScale(divisor, (long)dividendScale - scale);
5248         int raise = newScale - divisorScale;
5249         BigInteger scaledDivisor = bigMultiplyPowerTen(divisor, raise);
5250         return divideAndRound(dividend, scaledDivisor, scale, roundingMode, scale);
5251     }
5252 }
5253
5254 }

```

1.2 BigInteger.java

Secuencia 3: java/math/BigInteger.java

```

1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * Portions Copyright (c) 1995 Colin Plumb. All rights reserved.
28 */
29
30 package java.math;
31
32 import java.io.IOException;
33 import java.io.ObjectInputStream;
34 import java.io.ObjectOutputStream;
35 import java.io.ObjectStreamField;

```

```

36 import java.util.Arrays;
37 import java.util.Random;
38 import java.util.concurrent.ThreadLocalRandom;
39 import sun.misc.DoubleConsts;
40 import sun.misc.FloatConsts;
41
42 /**
43  * Immutable arbitrary-precision integers. All operations behave as if
44  * BigIntegers were represented in two's-complement notation (like Java's
45  * primitive integer types). BigInteger provides analogues to all of Java's
46  * primitive integer operators, and all relevant methods from java.lang.Math.
47  * Additionally, BigInteger provides operations for modular arithmetic, GCD
48  * calculation, primality testing, prime generation, bit manipulation,
49  * and a few other miscellaneous operations.
50  *
51  * <p>Semantics of arithmetic operations exactly mimic those of Java's integer
52  * arithmetic operators, as defined in <i>The Java Language Specification</i>.
53  * For example, division by zero throws an {@code ArithmeticException}, and
54  * division of a negative by a positive yields a negative (or zero) remainder.
55  * All of the details in the Spec concerning overflow are ignored, as
56  * BigIntegers are made as large as necessary to accommodate the results of an
57  * operation.
58  *
59  * <p>Semantics of shift operations extend those of Java's shift operators
60  * to allow for negative shift distances. A right-shift with a negative
61  * shift distance results in a left shift, and vice-versa. The unsigned
62  * right shift operator ({@code >>>}) is omitted, as this operation makes
63  * little sense in combination with the "infinite word size" abstraction
64  * provided by this class.
65  *
66  * <p>Semantics of bitwise logical operations exactly mimic those of Java's
67  * bitwise integer operators. The binary operators ({@code and},
68  * {@code or}, {@code xor}) implicitly perform sign extension on the shorter
69  * of the two operands prior to performing the operation.
70  *
71  * <p>Comparison operations perform signed integer comparisons, analogous to
72  * those performed by Java's relational and equality operators.
73  *
74  * <p>Modular arithmetic operations are provided to compute residues, perform
75  * exponentiation, and compute multiplicative inverses. These methods always
76  * return a non-negative result, between {@code 0} and {@code (modulus - 1)},
77  * inclusive.
78  *
79  * <p>Bit operations operate on a single bit of the two's-complement
80  * representation of their operand. If necessary, the operand is sign-
81  * extended so that it contains the designated bit. None of the single-bit
82  * operations can produce a BigInteger with a different sign from the
83  * BigInteger being operated on, as they affect only a single bit, and the
84  * "infinite word size" abstraction provided by this class ensures that there
85  * are infinitely many "virtual sign bits" preceding each BigInteger.
86  *
87  * <p>For the sake of brevity and clarity, pseudo-code is used throughout the
88  * descriptions of BigInteger methods. The pseudo-code expression

```

```

89  * {@code (i + j)} is shorthand for "a BigInteger whose value is
90  * that of the BigInteger {@code i} plus that of the BigInteger {@code j}."
91  * The pseudo-code expression {@code (i == j)} is shorthand for
92  * "{@code true} if and only if the BigInteger {@code i} represents the same
93  * value as the BigInteger {@code j}." Other pseudo-code expressions are
94  * interpreted similarly.
95  *
96  * <p>All methods and constructors in this class throw
97  * {@code NullPointerException} when passed
98  * a null object reference for any input parameter.
99  *
100 * BigInteger must support values in the range
101 * -2<sup>{@code Integer.MAX_VALUE}</sup> (exclusive) to
102 * +2<sup>{@code Integer.MAX_VALUE}</sup> (exclusive)
103 * and may support values outside of that range.
104 *
105 * The range of probable prime values is limited and may be less than
106 * the full supported positive range of {@code BigInteger}.
107 * The range must be at least 1 to 2<sup>500000000</sup>.
108 *
109 * @implNote
110 * BigInteger constructors and operations throw {@code ArithmeticException} when
111 * the result is out of the supported range of
112 * -2<sup>{@code Integer.MAX_VALUE}</sup> (exclusive) to
113 * +2<sup>{@code Integer.MAX_VALUE}</sup> (exclusive).
114 *
115 * @see      BigDecimal
116 * @author    Josh Bloch
117 * @author    Michael McCloskey
118 * @author    Alan Eliassen
119 * @author    Timothy Bukt
120 * @since     JDK1.1
121 */
122
123 public class BigInteger extends Number implements Comparable<BigInteger> {
124     /**
125      * The signum of this BigInteger: -1 for negative, 0 for zero, or
126      * 1 for positive. Note that the BigInteger zero <i>must</i> have
127      * a signum of 0. This is necessary to ensure that there is exactly one
128      * representation for each BigInteger value.
129      *
130      * @serial
131      */
132     final int signum;
133
134     /**
135      * The magnitude of this BigInteger, in <i>big-endian</i> order: the
136      * zeroth element of this array is the most-significant int of the
137      * magnitude. The magnitude must be "minimal" in that the most-significant
138      * int ({@code mag[0]}) must be non-zero. This is necessary to
139      * ensure that there is exactly one representation for each BigInteger
140      * value. Note that this implies that the BigInteger zero has a
141      * zero-length mag array.

```

```

142     */
143     final int[] mag;
144
145     // These "redundant fields" are initialized with recognizable nonsense
146     // values, and cached the first time they are needed (or never, if they
147     // aren't needed).
148
149     /**
150      * One plus the bitCount of this BigInteger. Zeros means uninitialized.
151      *
152      * @serial
153      * @see #bitCount
154      * @deprecated Deprecated since logical value is offset from stored
155      * value and correction factor is applied in accessor method.
156      */
157     @Deprecated
158     private int bitCount;
159
160     /**
161      * One plus the bitLength of this BigInteger. Zeros means uninitialized.
162      * (either value is acceptable).
163      *
164      * @serial
165      * @see #bitLength()
166      * @deprecated Deprecated since logical value is offset from stored
167      * value and correction factor is applied in accessor method.
168      */
169     @Deprecated
170     private int bitLength;
171
172     /**
173      * Two plus the lowest set bit of this BigInteger, as returned by
174      * getLowestSetBit().
175      *
176      * @serial
177      * @see #getLowestSetBit
178      * @deprecated Deprecated since logical value is offset from stored
179      * value and correction factor is applied in accessor method.
180      */
181     @Deprecated
182     private int lowestSetBit;
183
184     /**
185      * Two plus the index of the lowest-order int in the magnitude of this
186      * BigInteger that contains a nonzero int, or -2 (either value is acceptable).
187      * The least significant int has int-number 0, the next int in order of
188      * increasing significance has int-number 1, and so forth.
189      * @deprecated Deprecated since logical value is offset from stored
190      * value and correction factor is applied in accessor method.
191      */
192     @Deprecated
193     private int firstNonzeroIntNum;
194

```



```
195  /**
196   * This mask is used to obtain the value of an int as if it were unsigned.
197   */
198  final static long LONG_MASK = 0xffffffffL;
199
200  /**
201   * This constant limits {@code mag.length} of BigIntegers to the supported
202   * range.
203   */
204  private static final int MAX_MAG_LENGTH = Integer.MAX_VALUE / Integer.SIZE + 1; // (1 << 26)
205
206  /**
207   * Bit lengths larger than this constant can cause overflow in searchLen
208   * calculation and in BitSieve.singleSearch method.
209   */
210  private static final int PRIME_SEARCH_BIT_LENGTH_LIMIT = 500000000;
211
212  /**
213   * The threshold value for using Karatsuba multiplication. If the number
214   * of ints in both mag arrays are greater than this number, then
215   * Karatsuba multiplication will be used. This value is found
216   * experimentally to work well.
217   */
218  private static final int KARATSUBA_THRESHOLD = 80;
219
220  /**
221   * The threshold value for using 3-way Toom-Cook multiplication.
222   * If the number of ints in each mag array is greater than the
223   * Karatsuba threshold, and the number of ints in at least one of
224   * the mag arrays is greater than this threshold, then Toom-Cook
225   * multiplication will be used.
226   */
227  private static final int TOOM_COOK_THRESHOLD = 240;
228
229  /**
230   * The threshold value for using Karatsuba squaring. If the number
231   * of ints in the number are larger than this value,
232   * Karatsuba squaring will be used. This value is found
233   * experimentally to work well.
234   */
235  private static final int KARATSUBA_SQUARE_THRESHOLD = 128;
236
237  /**
238   * The threshold value for using Toom-Cook squaring. If the number
239   * of ints in the number are larger than this value,
240   * Toom-Cook squaring will be used. This value is found
241   * experimentally to work well.
242   */
243  private static final int TOOM_COOK_SQUARE_THRESHOLD = 216;
244
245  /**
246   * The threshold value for using Burnikel-Ziegler division. If the number
247   * of ints in the divisor are larger than this value, Burnikel-Ziegler
```

```

248     * division may be used. This value is found experimentally to work well.
249     */
250     static final int BURNIKEL_ZIEGLER_THRESHOLD = 80;
251
252     /**
253     * The offset value for using Burnikel-Ziegler division. If the number
254     * of ints in the divisor exceeds the Burnikel-Ziegler threshold, and the
255     * number of ints in the dividend is greater than the number of ints in the
256     * divisor plus this value, Burnikel-Ziegler division will be used. This
257     * value is found experimentally to work well.
258     */
259     static final int BURNIKEL_ZIEGLER_OFFSET = 40;
260
261     /**
262     * The threshold value for using Schoenhage recursive base conversion. If
263     * the number of ints in the number are larger than this value,
264     * the Schoenhage algorithm will be used. In practice, it appears that the
265     * Schoenhage routine is faster for any threshold down to 2, and is
266     * relatively flat for thresholds between 2-25, so this choice may be
267     * varied within this range for very small effect.
268     */
269     private static final int SCHOENHAGE_BASE_CONVERSION_THRESHOLD = 20;
270
271     /**
272     * The threshold value for using squaring code to perform multiplication
273     * of a {@code BigInteger} instance by itself. If the number of ints in
274     * the number are larger than this value, {@code multiply(this)} will
275     * return {@code square()}.
276     */
277     private static final int MULTIPLY_SQUARE_THRESHOLD = 20;
278
279     /**
280     * The threshold for using an intrinsic version of
281     * implMontgomeryXXX to perform Montgomery multiplication. If the
282     * number of ints in the number is more than this value we do not
283     * use the intrinsic.
284     */
285     private static final int MONTGOMERY_INTRINSIC_THRESHOLD = 512;
286
287
288     // Constructors
289
290     /**
291     * Translates a byte array containing the two's-complement binary
292     * representation of a BigInteger into a BigInteger. The input array is
293     * assumed to be in <i>big-endian</i> byte-order: the most significant
294     * byte is in the zeroth element.
295     *
296     * @param val big-endian two's-complement binary representation of
297     *             BigInteger.
298     * @throws NumberFormatException {@code val} is zero bytes long.
299     */
300     public BigInteger(byte[] val) {

```

```

301     if (val.length == 0)
302         throw new NumberFormatException("Zero length BigInteger");
303
304     if (val[0] < 0) {
305         mag = makePositive(val);
306         signum = -1;
307     } else {
308         mag = stripLeadingZeroBytes(val);
309         signum = (mag.length == 0 ? 0 : 1);
310     }
311     if (mag.length >= MAX_MAG_LENGTH) {
312         checkRange();
313     }
314 }
315
316 /**
317  * This private constructor translates an int array containing the
318  * two's-complement binary representation of a BigInteger into a
319  * BigInteger. The input array is assumed to be in <i>big-endian</i>
320  * int-order: the most significant int is in the zeroth element.
321  */
322 private BigInteger(int[] val) {
323     if (val.length == 0)
324         throw new NumberFormatException("Zero length BigInteger");
325
326     if (val[0] < 0) {
327         mag = makePositive(val);
328         signum = -1;
329     } else {
330         mag = trustedStripLeadingZeroInts(val);
331         signum = (mag.length == 0 ? 0 : 1);
332     }
333     if (mag.length >= MAX_MAG_LENGTH) {
334         checkRange();
335     }
336 }
337
338 /**
339  * Translates the sign-magnitude representation of a BigInteger into a
340  * BigInteger. The sign is represented as an integer signum value: -1 for
341  * negative, 0 for zero, or 1 for positive. The magnitude is a byte array
342  * in <i>big-endian</i> byte-order: the most significant byte is in the
343  * zeroth element. A zero-length magnitude array is permissible, and will
344  * result in a BigInteger value of 0, whether signum is -1, 0 or 1.
345  *
346  * @param signum signum of the number (-1 for negative, 0 for zero, 1
347  *             for positive).
348  * @param magnitude big-endian binary representation of the magnitude of
349  *                 the number.
350  * @throws NumberFormatException {@code signum} is not one of the three
351  *             legal values (-1, 0, and 1), or {@code signum} is 0 and
352  *             {@code magnitude} contains one or more non-zero bytes.
353  */

```

```

354 public BigInteger(int signum, byte[] magnitude) {
355     this.mag = stripLeadingZeroBytes(magnitude);
356
357     if (signum < -1 || signum > 1)
358         throw(new NumberFormatException("Invalid signum value"));
359
360     if (this.mag.length == 0) {
361         this.signum = 0;
362     } else {
363         if (signum == 0)
364             throw(new NumberFormatException("signum-magnitude mismatch"));
365         this.signum = signum;
366     }
367     if (mag.length >= MAX_MAG_LENGTH) {
368         checkRange();
369     }
370 }
371
372 /**
373  * A constructor for internal use that translates the sign-magnitude
374  * representation of a BigInteger into a BigInteger. It checks the
375  * arguments and copies the magnitude so this constructor would be
376  * safe for external use.
377  */
378 private BigInteger(int signum, int[] magnitude) {
379     this.mag = stripLeadingZeroInts(magnitude);
380
381     if (signum < -1 || signum > 1)
382         throw(new NumberFormatException("Invalid signum value"));
383
384     if (this.mag.length == 0) {
385         this.signum = 0;
386     } else {
387         if (signum == 0)
388             throw(new NumberFormatException("signum-magnitude mismatch"));
389         this.signum = signum;
390     }
391     if (mag.length >= MAX_MAG_LENGTH) {
392         checkRange();
393     }
394 }
395
396 /**
397  * Translates the String representation of a BigInteger in the
398  * specified radix into a BigInteger. The String representation
399  * consists of an optional minus or plus sign followed by a
400  * sequence of one or more digits in the specified radix. The
401  * character-to-digit mapping is provided by {@code
402  * Character.digit}. The String may not contain any extraneous
403  * characters (whitespace, for example).
404  *
405  * @param val String representation of BigInteger.
406  * @param radix radix to be used in interpreting {@code val}.

```

```

407 * @throws NumberFormatException {@code val} is not a valid representation
408 *       of a BigInteger in the specified radix, or {@code radix} is
409 *       outside the range from {@link Character#MIN_RADIX} to
410 *       {@link Character#MAX_RADIX}, inclusive.
411 * @see   Character#digit
412 */
413 public BigInteger(String val, int radix) {
414     int cursor = 0, numDigits;
415     final int len = val.length();
416
417     if (radix < Character.MIN_RADIX || radix > Character.MAX_RADIX)
418         throw new NumberFormatException("Radix out of range");
419     if (len == 0)
420         throw new NumberFormatException("Zero length BigInteger");
421
422     // Check for at most one leading sign
423     int sign = 1;
424     int index1 = val.lastIndexOf('-');
425     int index2 = val.lastIndexOf('+');
426     if (index1 >= 0) {
427         if (index1 != 0 || index2 >= 0) {
428             throw new NumberFormatException("Illegal embedded sign character");
429         }
430         sign = -1;
431         cursor = 1;
432     } else if (index2 >= 0) {
433         if (index2 != 0) {
434             throw new NumberFormatException("Illegal embedded sign character");
435         }
436         cursor = 1;
437     }
438     if (cursor == len)
439         throw new NumberFormatException("Zero length BigInteger");
440
441     // Skip leading zeros and compute number of digits in magnitude
442     while (cursor < len &&
443         Character.digit(val.charAt(cursor), radix) == 0) {
444         cursor++;
445     }
446
447     if (cursor == len) {
448         signum = 0;
449         mag = ZERO.mag;
450         return;
451     }
452
453     numDigits = len - cursor;
454     signum = sign;
455
456     // Pre-allocate array of expected size. May be too large but can
457     // never be too small. Typically exact.
458     long numBits = ((numDigits * bitsPerDigit[radix]) >>> 10) + 1;
459     if (numBits + 31 >= (1L << 32)) {

```

```

460         reportOverflow();
461     }
462     int numWords = (int) (numBits + 31) >>> 5;
463     int[] magnitude = new int[numWords];
464
465     // Process first (potentially short) digit group
466     int firstGroupLen = numDigits % digitsPerInt[radix];
467     if (firstGroupLen == 0)
468         firstGroupLen = digitsPerInt[radix];
469     String group = val.substring(cursor, cursor += firstGroupLen);
470     magnitude[numWords - 1] = Integer.parseInt(group, radix);
471     if (magnitude[numWords - 1] < 0)
472         throw new NumberFormatException("Illegal digit");
473
474     // Process remaining digit groups
475     int superRadix = intRadix[radix];
476     int groupVal = 0;
477     while (cursor < len) {
478         group = val.substring(cursor, cursor += digitsPerInt[radix]);
479         groupVal = Integer.parseInt(group, radix);
480         if (groupVal < 0)
481             throw new NumberFormatException("Illegal digit");
482         destructiveMulAdd(magnitude, superRadix, groupVal);
483     }
484     // Required for cases where the array was overallocated.
485     mag = trustedStripLeadingZeroInts(magnitude);
486     if (mag.length >= MAX_MAG_LENGTH) {
487         checkRange();
488     }
489 }
490
491 /*
492  * Constructs a new BigInteger using a char array with radix=10.
493  * Sign is precalculated outside and not allowed in the val.
494  */
495 BigInteger(char[] val, int sign, int len) {
496     int cursor = 0, numDigits;
497
498     // Skip leading zeros and compute number of digits in magnitude
499     while (cursor < len && Character.digit(val[cursor], 10) == 0) {
500         cursor++;
501     }
502     if (cursor == len) {
503         signum = 0;
504         mag = ZERO.mag;
505         return;
506     }
507
508     numDigits = len - cursor;
509     signum = sign;
510     // Pre-allocate array of expected size
511     int numWords;
512     if (len < 10) {

```

```

513         numWords = 1;
514     } else {
515         long numBits = ((numDigits * bitsPerDigit[10]) >>> 10) + 1;
516         if (numBits + 31 >= (1L << 32)) {
517             reportOverflow();
518         }
519         numWords = (int) (numBits + 31) >>> 5;
520     }
521     int[] magnitude = new int[numWords];
522
523     // Process first (potentially short) digit group
524     int firstGroupLen = numDigits % digitsPerInt[10];
525     if (firstGroupLen == 0)
526         firstGroupLen = digitsPerInt[10];
527     magnitude[numWords - 1] = parseInt(val, cursor, cursor += firstGroupLen);
528
529     // Process remaining digit groups
530     while (cursor < len) {
531         int groupVal = parseInt(val, cursor, cursor += digitsPerInt[10]);
532         destructiveMulAdd(magnitude, intRadix[10], groupVal);
533     }
534     mag = trustedStripLeadingZeroInts(magnitude);
535     if (mag.length >= MAX_MAG_LENGTH) {
536         checkRange();
537     }
538 }
539
540 // Create an integer with the digits between the two indexes
541 // Assumes start < end. The result may be negative, but it
542 // is to be treated as an unsigned value.
543 private int parseInt(char[] source, int start, int end) {
544     int result = Character.digit(source[start++], 10);
545     if (result == -1)
546         throw new NumberFormatException(new String(source));
547
548     for (int index = start; index < end; index++) {
549         int nextVal = Character.digit(source[index], 10);
550         if (nextVal == -1)
551             throw new NumberFormatException(new String(source));
552         result = 10*result + nextVal;
553     }
554
555     return result;
556 }
557
558 // bitsPerDigit in the given radix times 1024
559 // Rounded up to avoid underallocation.
560 private static long bitsPerDigit[] = { 0, 0,
561     1024, 1624, 2048, 2378, 2648, 2875, 3072, 3247, 3402, 3543, 3672,
562     3790, 3899, 4001, 4096, 4186, 4271, 4350, 4426, 4498, 4567, 4633,
563     4696, 4756, 4814, 4870, 4923, 4975, 5025, 5074, 5120, 5166, 5210,
564     5253, 5295};
565

```

```

566 // Multiply x array times word y in place, and add word z
567 private static void destructiveMulAdd(int[] x, int y, int z) {
568     // Perform the multiplication word by word
569     long ylong = y & LONG_MASK;
570     long zlong = z & LONG_MASK;
571     int len = x.length;
572
573     long product = 0;
574     long carry = 0;
575     for (int i = len-1; i >= 0; i--) {
576         product = ylong * (x[i] & LONG_MASK) + carry;
577         x[i] = (int)product;
578         carry = product >>> 32;
579     }
580
581     // Perform the addition
582     long sum = (x[len-1] & LONG_MASK) + zlong;
583     x[len-1] = (int)sum;
584     carry = sum >>> 32;
585     for (int i = len-2; i >= 0; i--) {
586         sum = (x[i] & LONG_MASK) + carry;
587         x[i] = (int)sum;
588         carry = sum >>> 32;
589     }
590 }
591
592 /**
593  * Translates the decimal String representation of a BigInteger into a
594  * BigInteger. The String representation consists of an optional minus
595  * sign followed by a sequence of one or more decimal digits. The
596  * character-to-digit mapping is provided by {@code Character.digit}.
597  * The String may not contain any extraneous characters (whitespace, for
598  * example).
599  *
600  * @param val decimal String representation of BigInteger.
601  * @throws NumberFormatException {@code val} is not a valid representation
602  *         of a BigInteger.
603  * @see Character#digit
604  */
605 public BigInteger(String val) {
606     this(val, 10);
607 }
608
609 /**
610  * Constructs a randomly generated BigInteger, uniformly distributed over
611  * the range 0 to (2<sup>{@code numBits}</sup> - 1), inclusive.
612  * The uniformity of the distribution assumes that a fair source of random
613  * bits is provided in {@code rnd}. Note that this constructor always
614  * constructs a non-negative BigInteger.
615  *
616  * @param numBits maximum bitLength of the new BigInteger.
617  * @param rnd source of randomness to be used in computing the new
618  *         BigInteger.

```



```

619     * @throws IllegalArgumentException {@code numBits} is negative.
620     * @see #bitLength()
621     */
622     public BigInteger(int numBits, Random rnd) {
623         this(1, randomBits(numBits, rnd));
624     }
625
626     private static byte[] randomBits(int numBits, Random rnd) {
627         if (numBits < 0)
628             throw new IllegalArgumentException("numBits must be non-negative");
629         int numBytes = (int)(((long)numBits+7)/8); // avoid overflow
630         byte[] randomBits = new byte[numBytes];
631
632         // Generate random bytes and mask out any excess bits
633         if (numBytes > 0) {
634             rnd.nextBytes(randomBits);
635             int excessBits = 8*numBytes - numBits;
636             randomBits[0] &= (1 << (8-excessBits)) - 1;
637         }
638         return randomBits;
639     }
640
641     /**
642     * Constructs a randomly generated positive BigInteger that is probably
643     * prime, with the specified bitLength.
644     *
645     * <p>It is recommended that the {@link #probablePrime probablePrime}
646     * method be used in preference to this constructor unless there
647     * is a compelling need to specify a certainty.
648     *
649     * @param bitLength bitLength of the returned BigInteger.
650     * @param certainty a measure of the uncertainty that the caller is
651     *     willing to tolerate. The probability that the new BigInteger
652     *     represents a prime number will exceed
653     *      $(1 - 1/2^{\text{certainty}})$ . The execution time of
654     *     this constructor is proportional to the value of this parameter.
655     * @param rnd source of random bits used to select candidates to be
656     *     tested for primality.
657     * @throws ArithmeticException {@code bitLength < 2} or {@code bitLength} is too large.
658     * @see #bitLength()
659     */
660     public BigInteger(int bitLength, int certainty, Random rnd) {
661         BigInteger prime;
662
663         if (bitLength < 2)
664             throw new ArithmeticException("bitLength < 2");
665         prime = (bitLength < SMALL_PRIME_THRESHOLD
666             ? smallPrime(bitLength, certainty, rnd)
667             : largePrime(bitLength, certainty, rnd));
668         signum = 1;
669         mag = prime.mag;
670     }
671

```

```

672 // Minimum size in bits that the requested prime number has
673 // before we use the large prime number generating algorithms.
674 // The cutoff of 95 was chosen empirically for best performance.
675 private static final int SMALL_PRIME_THRESHOLD = 95;
676
677 // Certainty required to meet the spec of probablePrime
678 private static final int DEFAULT_PRIME_CERTAINTY = 100;
679
680 /**
681  * Returns a positive BigInteger that is probably prime, with the
682  * specified bitLength. The probability that a BigInteger returned
683  * by this method is composite does not exceed  $2^{-100}$ .
684  *
685  * @param bitLength bitLength of the returned BigInteger.
686  * @param rnd source of random bits used to select candidates to be
687  *           tested for primality.
688  * @return a BigInteger of {code bitLength} bits that is probably prime
689  * @throws ArithmeticException {code bitLength < 2} or {code bitLength} is too large.
690  * @see #bitLength()
691  * @since 1.4
692  */
693 public static BigInteger probablePrime(int bitLength, Random rnd) {
694     if (bitLength < 2)
695         throw new ArithmeticException("bitLength < 2");
696
697     return (bitLength < SMALL_PRIME_THRESHOLD ?
698         smallPrime(bitLength, DEFAULT_PRIME_CERTAINTY, rnd) :
699         largePrime(bitLength, DEFAULT_PRIME_CERTAINTY, rnd));
700 }
701
702 /**
703  * Find a random number of the specified bitLength that is probably prime.
704  * This method is used for smaller primes, its performance degrades on
705  * larger bitlengths.
706  *
707  * This method assumes bitLength > 1.
708  */
709 private static BigInteger smallPrime(int bitLength, int certainty, Random rnd) {
710     int magLen = (bitLength + 31) >>> 5;
711     int temp[] = new int[magLen];
712     int highBit = 1 << ((bitLength+31) & 0x1f); // High bit of high int
713     int highMask = (highBit << 1) - 1; // Bits to keep in high int
714
715     while (true) {
716         // Construct a candidate
717         for (int i=0; i < magLen; i++)
718             temp[i] = rnd.nextInt();
719         temp[0] = (temp[0] & highMask) | highBit; // Ensure exact length
720         if (bitLength > 2)
721             temp[magLen-1] |= 1; // Make odd if bitlen > 2
722
723         BigInteger p = new BigInteger(temp, 1);
724

```

```

725     // Do cheap "pre-test" if applicable
726     if (bitLength > 6) {
727         long r = p.remainder(SMALL_PRIME_PRODUCT).longValue();
728         if ((r%3==0) || (r%5==0) || (r%7==0) || (r%11==0) ||
729             (r%13==0) || (r%17==0) || (r%19==0) || (r%23==0) ||
730             (r%29==0) || (r%31==0) || (r%37==0) || (r%41==0))
731             continue; // Candidate is composite; try another
732     }
733
734     // All candidates of bitLength 2 and 3 are prime by this point
735     if (bitLength < 4)
736         return p;
737
738     // Do expensive test if we survive pre-test (or it's inapplicable)
739     if (p.primeToCertainty(certainty, rnd))
740         return p;
741     }
742 }
743
744 private static final BigInteger SMALL_PRIME_PRODUCT
745     = valueOf(3L*5*7*11*13*17*19*23*29*31*37*41);
746
747 /**
748  * Find a random number of the specified bitLength that is probably prime.
749  * This method is more appropriate for larger bitlengths since it uses
750  * a sieve to eliminate most composites before using a more expensive
751  * test.
752  */
753 private static BigInteger largePrime(int bitLength, int certainty, Random rnd) {
754     BigInteger p;
755     p = new BigInteger(bitLength, rnd).setBit(bitLength-1);
756     p.mag[p.mag.length-1] &= 0xffffffff;
757
758     // Use a sieve length likely to contain the next prime number
759     int searchLen = getPrimeSearchLen(bitLength);
760     BitSieve searchSieve = new BitSieve(p, searchLen);
761     BigInteger candidate = searchSieve.retrieve(p, certainty, rnd);
762
763     while ((candidate == null) || (candidate.bitLength() != bitLength)) {
764         p = p.add(BigInteger.valueOf(2*searchLen));
765         if (p.bitLength() != bitLength)
766             p = new BigInteger(bitLength, rnd).setBit(bitLength-1);
767         p.mag[p.mag.length-1] &= 0xffffffff;
768         searchSieve = new BitSieve(p, searchLen);
769         candidate = searchSieve.retrieve(p, certainty, rnd);
770     }
771     return candidate;
772 }
773
774 /**
775  * Returns the first integer greater than this {@code BigInteger} that
776  * is probably prime. The probability that the number returned by this
777  * method is composite does not exceed  $2^{-100}$ . This method will

```

```

778 * never skip over a prime when searching: if it returns {@code p}, there
779 * is no prime {@code q} such that {@code this < q < p}.
780 *
781 * @return the first integer greater than this {@code BigInteger} that
782 *         is probably prime.
783 * @throws ArithmeticException {@code this < 0} or {@code this} is too large.
784 * @since 1.5
785 */
786 public BigInteger nextProbablePrime() {
787     if (this.signum < 0)
788         throw new ArithmeticException("start < 0: " + this);
789
790     // Handle trivial cases
791     if ((this.signum == 0) || this.equals(ONE))
792         return TWO;
793
794     BigInteger result = this.add(ONE);
795
796     // Fastpath for small numbers
797     if (result.bitLength() < SMALL_PRIME_THRESHOLD) {
798
799         // Ensure an odd number
800         if (!result.testBit(0))
801             result = result.add(ONE);
802
803         while (true) {
804             // Do cheap "pre-test" if applicable
805             if (result.bitLength() > 6) {
806                 long r = result.remainder(SMALL_PRIME_PRODUCT).longValue();
807                 if ((r%3==0) || (r%5==0) || (r%7==0) || (r%11==0) ||
808                     (r%13==0) || (r%17==0) || (r%19==0) || (r%23==0) ||
809                     (r%29==0) || (r%31==0) || (r%37==0) || (r%41==0)) {
810                     result = result.add(TWO);
811                     continue; // Candidate is composite; try another
812                 }
813             }
814
815             // All candidates of bitLength 2 and 3 are prime by this point
816             if (result.bitLength() < 4)
817                 return result;
818
819             // The expensive test
820             if (result.primeToCertainty(DEFAULT_PRIME_CERTAINTY, null))
821                 return result;
822
823             result = result.add(TWO);
824         }
825     }
826
827     // Start at previous even number
828     if (result.testBit(0))
829         result = result.subtract(ONE);
830

```

```

831     // Looking for the next large prime
832     int searchLen = getPrimeSearchLen(result.bitLength());
833
834     while (true) {
835         BitSieve searchSieve = new BitSieve(result, searchLen);
836         BigInteger candidate = searchSieve.retrieve(result,
837             DEFAULT_PRIME_CERTAINTY, null);
838         if (candidate != null)
839             return candidate;
840         result = result.add(BigInteger.valueOf(2 * searchLen));
841     }
842 }
843
844 private static int getPrimeSearchLen(int bitLength) {
845     if (bitLength > PRIME_SEARCH_BIT_LENGTH_LIMIT + 1) {
846         throw new ArithmeticException("Prime search implementation restriction on bitLength");
847     }
848     return bitLength / 20 * 64;
849 }
850
851 /**
852  * Returns {@code true} if this BigInteger is probably prime,
853  * {@code false} if it's definitely composite.
854  *
855  * This method assumes bitLength > 2.
856  *
857  * @param certainty a measure of the uncertainty that the caller is
858  *     willing to tolerate: if the call returns {@code true}
859  *     the probability that this BigInteger is prime exceeds
860  *     {@code (1 - 1/2<sup>certainty</sup>)}. The execution time of
861  *     this method is proportional to the value of this parameter.
862  * @return {@code true} if this BigInteger is probably prime,
863  *     {@code false} if it's definitely composite.
864  */
865 boolean primeToCertainty(int certainty, Random random) {
866     int rounds = 0;
867     int n = (Math.min(certainty, Integer.MAX_VALUE-1)+1)/2;
868
869     // The relationship between the certainty and the number of rounds
870     // we perform is given in the draft standard ANSI X9.80, "PRIME
871     // NUMBER GENERATION, PRIMALITY TESTING, AND PRIMALITY CERTIFICATES".
872     int sizeInBits = this.bitLength();
873     if (sizeInBits < 100) {
874         rounds = 50;
875         rounds = n < rounds ? n : rounds;
876         return passesMillerRabin(rounds, random);
877     }
878
879     if (sizeInBits < 256) {
880         rounds = 27;
881     } else if (sizeInBits < 512) {
882         rounds = 15;
883     } else if (sizeInBits < 768) {

```

```

884         rounds = 8;
885     } else if (sizeInBits < 1024) {
886         rounds = 4;
887     } else {
888         rounds = 2;
889     }
890     rounds = n < rounds ? n : rounds;
891
892     return passesMillerRabin(rounds, random) && passesLucasLehmer();
893 }
894
895 /**
896  * Returns true iff this BigInteger is a Lucas-Lehmer probable prime.
897  *
898  * The following assumptions are made:
899  * This BigInteger is a positive, odd number.
900  */
901 private boolean passesLucasLehmer() {
902     BigInteger thisPlusOne = this.add(ONE);
903
904     // Step 1
905     int d = 5;
906     while (jacobiSymbol(d, this) != -1) {
907         // 5, -7, 9, -11, ...
908         d = (d < 0) ? Math.abs(d)+2 : -(d+2);
909     }
910
911     // Step 2
912     BigInteger u = lucasLehmerSequence(d, thisPlusOne, this);
913
914     // Step 3
915     return u.mod(this).equals(ZERO);
916 }
917
918 /**
919  * Computes Jacobi(p,n).
920  * Assumes n positive, odd, n>=3.
921  */
922 private static int jacobiSymbol(int p, BigInteger n) {
923     if (p == 0)
924         return 0;
925
926     // Algorithm and comments adapted from Colin Plumb's C library.
927     int j = 1;
928     int u = n.mag[n.mag.length-1];
929
930     // Make p positive
931     if (p < 0) {
932         p = -p;
933         int n8 = u & 7;
934         if ((n8 == 3) || (n8 == 7))
935             j = -j; // 3 (011) or 7 (111) mod 8
936     }

```

```

937
938 // Get rid of factors of 2 in p
939 while ((p & 3) == 0)
940     p >>= 2;
941 if ((p & 1) == 0) {
942     p >>= 1;
943     if (((u ^ (u>>1)) & 2) != 0)
944         j = -j; // 3 (011) or 5 (101) mod 8
945 }
946 if (p == 1)
947     return j;
948 // Then, apply quadratic reciprocity
949 if ((p & u & 2) != 0) // p = u = 3 (mod 4)?
950     j = -j;
951 // And reduce u mod p
952 u = n.mod(BigInteger.valueOf(p)).intValue();
953
954 // Now compute Jacobi(u,p), u < p
955 while (u != 0) {
956     while ((u & 3) == 0)
957         u >>= 2;
958     if ((u & 1) == 0) {
959         u >>= 1;
960         if (((p ^ (p>>1)) & 2) != 0)
961             j = -j; // 3 (011) or 5 (101) mod 8
962     }
963     if (u == 1)
964         return j;
965     // Now both u and p are odd, so use quadratic reciprocity
966     assert (u < p);
967     int t = u; u = p; p = t;
968     if ((u & p & 2) != 0) // u = p = 3 (mod 4)?
969         j = -j;
970     // Now u >= p, so it can be reduced
971     u %= p;
972 }
973 return 0;
974 }
975
976 private static BigInteger lucasLehmerSequence(int z, BigInteger k, BigInteger n) {
977     BigInteger d = BigInteger.valueOf(z);
978     BigInteger u = ONE; BigInteger u2;
979     BigInteger v = ONE; BigInteger v2;
980
981     for (int i=k.bitLength()-2; i >= 0; i--) {
982         u2 = u.multiply(v).mod(n);
983
984         v2 = v.square().add(d.multiply(u.square())).mod(n);
985         if (v2.testBit(0))
986             v2 = v2.subtract(n);
987
988         v2 = v2.shiftRight(1);
989

```

```

990         u = u2; v = v2;
991         if (k.testBit(i)) {
992             u2 = u.add(v).mod(n);
993             if (u2.testBit(0))
994                 u2 = u2.subtract(n);
995
996             u2 = u2.shiftRight(1);
997             v2 = v.add(d.multiply(u)).mod(n);
998             if (v2.testBit(0))
999                 v2 = v2.subtract(n);
1000             v2 = v2.shiftRight(1);
1001
1002             u = u2; v = v2;
1003         }
1004     }
1005     return u;
1006 }
1007
1008 /**
1009  * Returns true iff this BigInteger passes the specified number of
1010  * Miller-Rabin tests. This test is taken from the DSA spec (NIST FIPS
1011  * 186-2).
1012  *
1013  * The following assumptions are made:
1014  * This BigInteger is a positive, odd number greater than 2.
1015  * iterations<=50.
1016  */
1017 private boolean passesMillerRabin(int iterations, Random rnd) {
1018     // Find a and m such that m is odd and this == 1 + 2**a * m
1019     BigInteger thisMinusOne = this.subtract(ONE);
1020     BigInteger m = thisMinusOne;
1021     int a = m.getLowestSetBit();
1022     m = m.shiftRight(a);
1023
1024     // Do the tests
1025     if (rnd == null) {
1026         rnd = ThreadLocalRandom.current();
1027     }
1028     for (int i=0; i < iterations; i++) {
1029         // Generate a uniform random on (1, this)
1030         BigInteger b;
1031         do {
1032             b = new BigInteger(this.bitLength(), rnd);
1033         } while (b.compareTo(ONE) <= 0 || b.compareTo(this) >= 0);
1034
1035         int j = 0;
1036         BigInteger z = b.modPow(m, this);
1037         while (!((j == 0 && z.equals(ONE)) || z.equals(thisMinusOne))) {
1038             if (j > 0 && z.equals(ONE) || ++j == a)
1039                 return false;
1040             z = z.modPow(TWO, this);
1041         }
1042     }

```



```

1043     return true;
1044 }
1045
1046 /**
1047  * This internal constructor differs from its public cousin
1048  * with the arguments reversed in two ways: it assumes that its
1049  * arguments are correct, and it doesn't copy the magnitude array.
1050  */
1051 BigInteger(int[] magnitude, int signum) {
1052     this.signum = (magnitude.length == 0 ? 0 : signum);
1053     this.mag = magnitude;
1054     if (mag.length >= MAX_MAG_LENGTH) {
1055         checkRange();
1056     }
1057 }
1058
1059 /**
1060  * This private constructor is for internal use and assumes that its
1061  * arguments are correct.
1062  */
1063 private BigInteger(byte[] magnitude, int signum) {
1064     this.signum = (magnitude.length == 0 ? 0 : signum);
1065     this.mag = stripLeadingZeroBytes(magnitude);
1066     if (mag.length >= MAX_MAG_LENGTH) {
1067         checkRange();
1068     }
1069 }
1070
1071 /**
1072  * Throws an {@code ArithmeticException} if the {@code BigInteger} would be
1073  * out of the supported range.
1074  *
1075  * @throws ArithmeticException if {@code this} exceeds the supported range.
1076  */
1077 private void checkRange() {
1078     if (mag.length > MAX_MAG_LENGTH || mag.length == MAX_MAG_LENGTH && mag[0] < 0) {
1079         reportOverflow();
1080     }
1081 }
1082
1083 private static void reportOverflow() {
1084     throw new ArithmeticException("BigInteger would overflow supported range");
1085 }
1086
1087 //Static Factory Methods
1088
1089 /**
1090  * Returns a BigInteger whose value is equal to that of the
1091  * specified {@code long}. This "static factory method" is
1092  * provided in preference to a ({@code long}) constructor
1093  * because it allows for reuse of frequently used BigIntegers.
1094  *
1095  * @param val value of the BigInteger to return.

```

```

1096     * @return a BigInteger with the specified value.
1097     */
1098     public static BigInteger valueOf(long val) {
1099         // If -MAX_CONSTANT < val < MAX_CONSTANT, return stashed constant
1100         if (val == 0)
1101             return ZERO;
1102         if (val > 0 && val <= MAX_CONSTANT)
1103             return posConst[(int) val];
1104         else if (val < 0 && val >= -MAX_CONSTANT)
1105             return negConst[(int) -val];
1106
1107         return new BigInteger(val);
1108     }
1109
1110     /**
1111     * Constructs a BigInteger with the specified value, which may not be zero.
1112     */
1113     private BigInteger(long val) {
1114         if (val < 0) {
1115             val = -val;
1116             signum = -1;
1117         } else {
1118             signum = 1;
1119         }
1120
1121         int highWord = (int)(val >>> 32);
1122         if (highWord == 0) {
1123             mag = new int[1];
1124             mag[0] = (int)val;
1125         } else {
1126             mag = new int[2];
1127             mag[0] = highWord;
1128             mag[1] = (int)val;
1129         }
1130     }
1131
1132     /**
1133     * Returns a BigInteger with the given two's complement representation.
1134     * Assumes that the input array will not be modified (the returned
1135     * BigInteger will reference the input array if feasible).
1136     */
1137     private static BigInteger valueOf(int val[]) {
1138         return (val[0] > 0 ? new BigInteger(val, 1) : new BigInteger(val));
1139     }
1140
1141     // Constants
1142
1143     /**
1144     * Initialize static constant array when class is loaded.
1145     */
1146     private final static int MAX_CONSTANT = 16;
1147     private static BigInteger posConst[] = new BigInteger[MAX_CONSTANT+1];
1148     private static BigInteger negConst[] = new BigInteger[MAX_CONSTANT+1];

```

```

1149
1150 /**
1151  * The cache of powers of each radix. This allows us to not have to
1152  * recalculate powers of radix^(2^n) more than once. This speeds
1153  * Schoenhage recursive base conversion significantly.
1154  */
1155 private static volatile BigInteger[] powerCache;
1156
1157 /** The cache of logarithms of radices for base conversion. */
1158 private static final double[] logCache;
1159
1160 /** The natural log of 2. This is used in computing cache indices. */
1161 private static final double LOG_TWO = Math.log(2.0);
1162
1163 static {
1164     for (int i = 1; i <= MAX_CONSTANT; i++) {
1165         int[] magnitude = new int[1];
1166         magnitude[0] = i;
1167         posConst[i] = new BigInteger(magnitude, 1);
1168         negConst[i] = new BigInteger(magnitude, -1);
1169     }
1170
1171     /**
1172      * Initialize the cache of radix^(2^x) values used for base conversion
1173      * with just the very first value. Additional values will be created
1174      * on demand.
1175      */
1176     powerCache = new BigInteger[Character.MAX_RADIX+1];
1177     logCache = new double[Character.MAX_RADIX+1];
1178
1179     for (int i=Character.MIN_RADIX; i <= Character.MAX_RADIX; i++) {
1180         powerCache[i] = new BigInteger[] { BigInteger.valueOf(i) };
1181         logCache[i] = Math.log(i);
1182     }
1183 }
1184
1185 /**
1186  * The BigInteger constant zero.
1187  *
1188  * @since 1.2
1189  */
1190 public static final BigInteger ZERO = new BigInteger(new int[0], 0);
1191
1192 /**
1193  * The BigInteger constant one.
1194  *
1195  * @since 1.2
1196  */
1197 public static final BigInteger ONE = valueOf(1);
1198
1199 /**
1200  * The BigInteger constant two. (Not exported.)
1201  */

```

```

1202     private static final BigInteger TWO = valueOf(2);
1203
1204     /**
1205      * The BigInteger constant -1. (Not exported.)
1206      */
1207     private static final BigInteger NEGATIVE_ONE = valueOf(-1);
1208
1209     /**
1210      * The BigInteger constant ten.
1211      *
1212      * @since 1.5
1213      */
1214     public static final BigInteger TEN = valueOf(10);
1215
1216     // Arithmetic Operations
1217
1218     /**
1219      * Returns a BigInteger whose value is {@code (this + val)}.
1220      *
1221      * @param val value to be added to this BigInteger.
1222      * @return {@code this + val}
1223      */
1224     public BigInteger add(BigInteger val) {
1225         if (val.signum == 0)
1226             return this;
1227         if (signum == 0)
1228             return val;
1229         if (val.signum == signum)
1230             return new BigInteger(add(mag, val.mag), signum);
1231
1232         int cmp = compareMagnitude(val);
1233         if (cmp == 0)
1234             return ZERO;
1235         int[] resultMag = (cmp > 0 ? subtract(mag, val.mag)
1236                                     : subtract(val.mag, mag));
1237         resultMag = trustedStripLeadingZeroInts(resultMag);
1238
1239         return new BigInteger(resultMag, cmp == signum ? 1 : -1);
1240     }
1241
1242     /**
1243      * Package private methods used by BigDecimal code to add a BigInteger
1244      * with a long. Assumes val is not equal to INFLATED.
1245      */
1246     BigInteger add(long val) {
1247         if (val == 0)
1248             return this;
1249         if (signum == 0)
1250             return valueOf(val);
1251         if (Long.signum(val) == signum)
1252             return new BigInteger(add(mag, Math.abs(val)), signum);
1253         int cmp = compareMagnitude(val);
1254         if (cmp == 0)

```

```

1255         return ZERO;
1256         int[] resultMag = (cmp > 0 ? subtract(mag, Math.abs(val)) : subtract(Math.abs(val), mag));
1257         resultMag = trustedStripLeadingZeroInts(resultMag);
1258         return new BigInteger(resultMag, cmp == signum ? 1 : -1);
1259     }
1260
1261     /**
1262      * Adds the contents of the int array x and long value val. This
1263      * method allocates a new int array to hold the answer and returns
1264      * a reference to that array. Assumes x.length > 0 and val is
1265      * non-negative
1266      */
1267     private static int[] add(int[] x, long val) {
1268         int[] y;
1269         long sum = 0;
1270         int xIndex = x.length;
1271         int[] result;
1272         int highWord = (int)(val >>> 32);
1273         if (highWord == 0) {
1274             result = new int[xIndex];
1275             sum = (x[--xIndex] & LONG_MASK) + val;
1276             result[xIndex] = (int)sum;
1277         } else {
1278             if (xIndex == 1) {
1279                 result = new int[2];
1280                 sum = val + (x[0] & LONG_MASK);
1281                 result[1] = (int)sum;
1282                 result[0] = (int)(sum >>> 32);
1283                 return result;
1284             } else {
1285                 result = new int[xIndex];
1286                 sum = (x[--xIndex] & LONG_MASK) + (val & LONG_MASK);
1287                 result[xIndex] = (int)sum;
1288                 sum = (x[--xIndex] & LONG_MASK) + (highWord & LONG_MASK) + (sum >>> 32);
1289                 result[xIndex] = (int)sum;
1290             }
1291         }
1292         // Copy remainder of longer number while carry propagation is required
1293         boolean carry = (sum >>> 32 != 0);
1294         while (xIndex > 0 && carry)
1295             carry = ((result[--xIndex] = x[xIndex] + 1) == 0);
1296         // Copy remainder of longer number
1297         while (xIndex > 0)
1298             result[--xIndex] = x[xIndex];
1299         // Grow result if necessary
1300         if (carry) {
1301             int bigger[] = new int[result.length + 1];
1302             System.arraycopy(result, 0, bigger, 1, result.length);
1303             bigger[0] = 0x01;
1304             return bigger;
1305         }
1306         return result;
1307     }

```

```

1308
1309 /**
1310  * Adds the contents of the int arrays x and y. This method allocates
1311  * a new int array to hold the answer and returns a reference to that
1312  * array.
1313  */
1314 private static int[] add(int[] x, int[] y) {
1315     // If x is shorter, swap the two arrays
1316     if (x.length < y.length) {
1317         int[] tmp = x;
1318         x = y;
1319         y = tmp;
1320     }
1321
1322     int xIndex = x.length;
1323     int yIndex = y.length;
1324     int result[] = new int[xIndex];
1325     long sum = 0;
1326     if (yIndex == 1) {
1327         sum = (x[--xIndex] & LONG_MASK) + (y[0] & LONG_MASK);
1328         result[xIndex] = (int)sum;
1329     } else {
1330         // Add common parts of both numbers
1331         while (yIndex > 0) {
1332             sum = (x[--xIndex] & LONG_MASK) +
1333                 (y[--yIndex] & LONG_MASK) + (sum >>> 32);
1334             result[xIndex] = (int)sum;
1335         }
1336     }
1337     // Copy remainder of longer number while carry propagation is required
1338     boolean carry = (sum >>> 32 != 0);
1339     while (xIndex > 0 && carry)
1340         carry = ((result[--xIndex] = x[xIndex] + 1) == 0);
1341
1342     // Copy remainder of longer number
1343     while (xIndex > 0)
1344         result[--xIndex] = x[xIndex];
1345
1346     // Grow result if necessary
1347     if (carry) {
1348         int bigger[] = new int[result.length + 1];
1349         System.arraycopy(result, 0, bigger, 1, result.length);
1350         bigger[0] = 0x01;
1351         return bigger;
1352     }
1353     return result;
1354 }
1355
1356 private static int[] subtract(long val, int[] little) {
1357     int highWord = (int)(val >>> 32);
1358     if (highWord == 0) {
1359         int result[] = new int[1];
1360         result[0] = (int)(val - (little[0] & LONG_MASK));

```

```

1361         return result;
1362     } else {
1363         int result[] = new int[2];
1364         if (little.length == 1) {
1365             long difference = ((int)val & LONG_MASK) - (little[0] & LONG_MASK);
1366             result[1] = (int)difference;
1367             // Subtract remainder of longer number while borrow propagates
1368             boolean borrow = (difference >> 32 != 0);
1369             if (borrow) {
1370                 result[0] = highWord - 1;
1371             } else { // Copy remainder of longer number
1372                 result[0] = highWord;
1373             }
1374             return result;
1375         } else { // little.length == 2
1376             long difference = ((int)val & LONG_MASK) - (little[1] & LONG_MASK);
1377             result[1] = (int)difference;
1378             difference = (highWord & LONG_MASK) - (little[0] & LONG_MASK) + (difference >> 32);
1379             result[0] = (int)difference;
1380             return result;
1381         }
1382     }
1383 }
1384
1385 /**
1386  * Subtracts the contents of the second argument (val) from the
1387  * first (big). The first int array (big) must represent a larger number
1388  * than the second. This method allocates the space necessary to hold the
1389  * answer.
1390  * assumes val >= 0
1391  */
1392 private static int[] subtract(int[] big, long val) {
1393     int highWord = (int)(val >>> 32);
1394     int bigIndex = big.length;
1395     int result[] = new int[bigIndex];
1396     long difference = 0;
1397
1398     if (highWord == 0) {
1399         difference = (big[--bigIndex] & LONG_MASK) - val;
1400         result[bigIndex] = (int)difference;
1401     } else {
1402         difference = (big[--bigIndex] & LONG_MASK) - (val & LONG_MASK);
1403         result[bigIndex] = (int)difference;
1404         difference = (big[--bigIndex] & LONG_MASK) - (highWord & LONG_MASK) + (difference >>
1405             32);
1406         result[bigIndex] = (int)difference;
1407     }
1408
1409     // Subtract remainder of longer number while borrow propagates
1410     boolean borrow = (difference >> 32 != 0);
1411     while (bigIndex > 0 && borrow)
1412         borrow = ((result[--bigIndex] = big[bigIndex] - 1) == -1);

```

```

1413     // Copy remainder of longer number
1414     while (bigIndex > 0)
1415         result[--bigIndex] = big[bigIndex];
1416
1417     return result;
1418 }
1419
1420 /**
1421  * Returns a BigInteger whose value is {@code (this - val)}.
1422  *
1423  * @param val value to be subtracted from this BigInteger.
1424  * @return {@code this - val}
1425  */
1426 public BigInteger subtract(BigInteger val) {
1427     if (val.signum == 0)
1428         return this;
1429     if (signum == 0)
1430         return val.negate();
1431     if (val.signum != signum)
1432         return new BigInteger(add(mag, val.mag), signum);
1433
1434     int cmp = compareMagnitude(val);
1435     if (cmp == 0)
1436         return ZERO;
1437     int[] resultMag = (cmp > 0 ? subtract(mag, val.mag)
1438                               : subtract(val.mag, mag));
1439     resultMag = trustedStripLeadingZeroInts(resultMag);
1440     return new BigInteger(resultMag, cmp == signum ? 1 : -1);
1441 }
1442
1443 /**
1444  * Subtracts the contents of the second int arrays (little) from the
1445  * first (big). The first int array (big) must represent a larger number
1446  * than the second. This method allocates the space necessary to hold the
1447  * answer.
1448  */
1449 private static int[] subtract(int[] big, int[] little) {
1450     int bigIndex = big.length;
1451     int result[] = new int[bigIndex];
1452     int littleIndex = little.length;
1453     long difference = 0;
1454
1455     // Subtract common parts of both numbers
1456     while (littleIndex > 0) {
1457         difference = (big[--bigIndex] & LONG_MASK) -
1458                     (little[--littleIndex] & LONG_MASK) +
1459                     (difference >> 32);
1460         result[bigIndex] = (int)difference;
1461     }
1462
1463     // Subtract remainder of longer number while borrow propagates
1464     boolean borrow = (difference >> 32 != 0);
1465     while (bigIndex > 0 && borrow)

```



```

1466         borrow = ((result[--bigIndex] = big[bigIndex] - 1) == -1);
1467
1468         // Copy remainder of longer number
1469         while (bigIndex > 0)
1470             result[--bigIndex] = big[bigIndex];
1471
1472         return result;
1473     }
1474
1475     /**
1476      * Returns a BigInteger whose value is {@code (this * val)}.
1477      *
1478      * @implNote An implementation may offer better algorithmic
1479      * performance when {@code val == this}.
1480      *
1481      * @param val value to be multiplied by this BigInteger.
1482      * @return {@code this * val}
1483      */
1484     public BigInteger multiply(BigInteger val) {
1485         if (val.signum == 0 || signum == 0)
1486             return ZERO;
1487
1488         int xlen = mag.length;
1489
1490         if (val == this && xlen > MULTIPLY_SQUARE_THRESHOLD) {
1491             return square();
1492         }
1493
1494         int ylen = val.mag.length;
1495
1496         if ((xlen < KARATSUBA_THRESHOLD) || (ylen < KARATSUBA_THRESHOLD)) {
1497             int resultSign = signum == val.signum ? 1 : -1;
1498             if (val.mag.length == 1) {
1499                 return multiplyByInt(mag, val.mag[0], resultSign);
1500             }
1501             if (mag.length == 1) {
1502                 return multiplyByInt(val.mag, mag[0], resultSign);
1503             }
1504             int[] result = multiplyToLen(mag, xlen,
1505                                         val.mag, ylen, null);
1506             result = trustedStripLeadingZeroInts(result);
1507             return new BigInteger(result, resultSign);
1508         } else {
1509             if ((xlen < TOOM_COOK_THRESHOLD) && (ylen < TOOM_COOK_THRESHOLD)) {
1510                 return multiplyKaratsuba(this, val);
1511             } else {
1512                 return multiplyToomCook3(this, val);
1513             }
1514         }
1515     }
1516
1517     private static BigInteger multiplyByInt(int[] x, int y, int sign) {
1518         if (Integer.bitCount(y) == 1) {

```

```

1519         return new BigInteger(shiftLeft(x,Integer.numberOfTrailingZeros(y)), sign);
1520     }
1521     int xlen = x.length;
1522     int[] rmag = new int[xlen + 1];
1523     long carry = 0;
1524     long y1 = y & LONG_MASK;
1525     int rstart = rmag.length - 1;
1526     for (int i = xlen - 1; i >= 0; i--) {
1527         long product = (x[i] & LONG_MASK) * y1 + carry;
1528         rmag[rstart--] = (int)product;
1529         carry = product >>> 32;
1530     }
1531     if (carry == 0L) {
1532         rmag = java.util.Arrays.copyOfRange(rmag, 1, rmag.length);
1533     } else {
1534         rmag[rstart] = (int)carry;
1535     }
1536     return new BigInteger(rmag, sign);
1537 }
1538
1539 /**
1540  * Package private methods used by BigDecimal code to multiply a BigInteger
1541  * with a long. Assumes v is not equal to INFLATED.
1542  */
1543 BigInteger multiply(long v) {
1544     if (v == 0 || signum == 0)
1545         return ZERO;
1546     if (v == BigDecimal.INFLATED)
1547         return multiply(BigInteger.valueOf(v));
1548     int rsign = (v > 0 ? signum : -signum);
1549     if (v < 0)
1550         v = -v;
1551     long dh = v >>> 32; // higher order bits
1552     long dl = v & LONG_MASK; // lower order bits
1553
1554     int xlen = mag.length;
1555     int[] value = mag;
1556     int[] rmag = (dh == 0L) ? (new int[xlen + 1]) : (new int[xlen + 2]);
1557     long carry = 0;
1558     int rstart = rmag.length - 1;
1559     for (int i = xlen - 1; i >= 0; i--) {
1560         long product = (value[i] & LONG_MASK) * dl + carry;
1561         rmag[rstart--] = (int)product;
1562         carry = product >>> 32;
1563     }
1564     rmag[rstart] = (int)carry;
1565     if (dh != 0L) {
1566         carry = 0;
1567         rstart = rmag.length - 2;
1568         for (int i = xlen - 1; i >= 0; i--) {
1569             long product = (value[i] & LONG_MASK) * dh +
1570                 (rmag[rstart] & LONG_MASK) + carry;
1571             rmag[rstart--] = (int)product;

```

```

1572         carry = product >>> 32;
1573     }
1574     rmag[0] = (int)carry;
1575 }
1576 if (carry == 0L)
1577     rmag = java.util.Arrays.copyOfRange(rmag, 1, rmag.length);
1578 return new BigInteger(rmag, rsign);
1579 }
1580
1581 /**
1582  * Multiplies int arrays x and y to the specified lengths and places
1583  * the result into z. There will be no leading zeros in the resultant array.
1584  */
1585 private static int[] multiplyToLen(int[] x, int xlen, int[] y, int ylen, int[] z) {
1586     int xstart = xlen - 1;
1587     int ystart = ylen - 1;
1588
1589     if (z == null || z.length < (xlen+ ylen))
1590         z = new int[xlen+ylen];
1591
1592     long carry = 0;
1593     for (int j=ystart, k=ystart+1+xstart; j >= 0; j--, k--) {
1594         long product = (y[j] & LONG_MASK) *
1595             (x[xstart] & LONG_MASK) + carry;
1596         z[k] = (int)product;
1597         carry = product >>> 32;
1598     }
1599     z[xstart] = (int)carry;
1600
1601     for (int i = xstart-1; i >= 0; i--) {
1602         carry = 0;
1603         for (int j=ystart, k=ystart+1+i; j >= 0; j--, k--) {
1604             long product = (y[j] & LONG_MASK) *
1605                 (x[i] & LONG_MASK) +
1606                 (z[k] & LONG_MASK) + carry;
1607             z[k] = (int)product;
1608             carry = product >>> 32;
1609         }
1610         z[i] = (int)carry;
1611     }
1612     return z;
1613 }
1614
1615 /**
1616  * Multiplies two BigIntegers using the Karatsuba multiplication
1617  * algorithm. This is a recursive divide-and-conquer algorithm which is
1618  * more efficient for large numbers than what is commonly called the
1619  * "grade-school" algorithm used in multiplyToLen. If the numbers to be
1620  * multiplied have length n, the "grade-school" algorithm has an
1621  * asymptotic complexity of  $O(n^2)$ . In contrast, the Karatsuba algorithm
1622  * has complexity of  $O(n^{\log_2(3)})$ , or  $O(n^{1.585})$ . It achieves this
1623  * increased performance by doing 3 multiplies instead of 4 when
1624  * evaluating the product. As it has some overhead, should be used when

```

```

1625     * both numbers are larger than a certain threshold (found
1626     * experimentally).
1627     *
1628     * See: http://en.wikipedia.org/wiki/Karatsuba\_algorithm
1629     */
1630     private static BigInteger multiplyKaratsuba(BigInteger x, BigInteger y) {
1631         int xlen = x.mag.length;
1632         int ylen = y.mag.length;
1633
1634         // The number of ints in each half of the number.
1635         int half = (Math.max(xlen, ylen)+1) / 2;
1636
1637         // xl and yl are the lower halves of x and y respectively,
1638         // xh and yh are the upper halves.
1639         BigInteger xl = x.getLower(half);
1640         BigInteger xh = x.getUpper(half);
1641         BigInteger yl = y.getLower(half);
1642         BigInteger yh = y.getUpper(half);
1643
1644         BigInteger p1 = xh.multiply(yh); // p1 = xh*yh
1645         BigInteger p2 = xl.multiply(yl); // p2 = xl*yl
1646
1647         // p3=(xh+xl)*(yh+yl)
1648         BigInteger p3 = xh.add(xl).multiply(yh.add(yl));
1649
1650         // result = p1 * 2^(32*2*half) + (p3 - p1 - p2) * 2^(32*half) + p2
1651         BigInteger result = p1.shiftLeft(32*half).add(p3.subtract(p1).subtract(p2)).shiftLeft(32*
            half).add(p2);
1652
1653         if (x.signum != y.signum) {
1654             return result.negate();
1655         } else {
1656             return result;
1657         }
1658     }
1659
1660     /**
1661     * Multiplies two BigIntegers using a 3-way Toom-Cook multiplication
1662     * algorithm. This is a recursive divide-and-conquer algorithm which is
1663     * more efficient for large numbers than what is commonly called the
1664     * "grade-school" algorithm used in multiplyToLen. If the numbers to be
1665     * multiplied have length n, the "grade-school" algorithm has an
1666     * asymptotic complexity of  $O(n^2)$ . In contrast, 3-way Toom-Cook has a
1667     * complexity of about  $O(n^{1.465})$ . It achieves this increased asymptotic
1668     * performance by breaking each number into three parts and by doing 5
1669     * multiplies instead of 9 when evaluating the product. Due to overhead
1670     * (additions, shifts, and one division) in the Toom-Cook algorithm, it
1671     * should only be used when both numbers are larger than a certain
1672     * threshold (found experimentally). This threshold is generally larger
1673     * than that for Karatsuba multiplication, so this algorithm is generally
1674     * only used when numbers become significantly larger.
1675     *
1676     * The algorithm used is the "optimal" 3-way Toom-Cook algorithm outlined

```

```

1677 * by Marco Bodrato.
1678 *
1679 * See: http://bodrato.it/toom-cook/
1680 *      http://bodrato.it/papers/#WAIFI2007
1681 *
1682 * "Towards Optimal Toom-Cook Multiplication for Univariate and
1683 * Multivariate Polynomials in Characteristic 2 and 0." by Marco BODRATO;
1684 * In C.Carlet and B.Sunar, Eds., "WAIFI'07 proceedings", p. 116-133,
1685 * LNCS #4547. Springer, Madrid, Spain, June 21-22, 2007.
1686 *
1687 */
1688 private static BigInteger multiplyToomCook3(BigInteger a, BigInteger b) {
1689     int alen = a.mag.length;
1690     int blen = b.mag.length;
1691
1692     int largest = Math.max(alen, blen);
1693
1694     // k is the size (in ints) of the lower-order slices.
1695     int k = (largest+2)/3; // Equal to ceil(largest/3)
1696
1697     // r is the size (in ints) of the highest-order slice.
1698     int r = largest - 2*k;
1699
1700     // Obtain slices of the numbers. a2 and b2 are the most significant
1701     // bits of the numbers a and b, and a0 and b0 the least significant.
1702     BigInteger a0, a1, a2, b0, b1, b2;
1703     a2 = a.getToomSlice(k, r, 0, largest);
1704     a1 = a.getToomSlice(k, r, 1, largest);
1705     a0 = a.getToomSlice(k, r, 2, largest);
1706     b2 = b.getToomSlice(k, r, 0, largest);
1707     b1 = b.getToomSlice(k, r, 1, largest);
1708     b0 = b.getToomSlice(k, r, 2, largest);
1709
1710     BigInteger v0, v1, v2, vm1, vinf, t1, t2, tm1, da1, db1;
1711
1712     v0 = a0.multiply(b0);
1713     da1 = a2.add(a0);
1714     db1 = b2.add(b0);
1715     vm1 = da1.subtract(a1).multiply(db1.subtract(b1));
1716     da1 = da1.add(a1);
1717     db1 = db1.add(b1);
1718     v1 = da1.multiply(db1);
1719     v2 = da1.add(a2).shiftLeft(1).subtract(a0).multiply(
1720         db1.add(b2).shiftLeft(1).subtract(b0));
1721     vinf = a2.multiply(b2);
1722
1723     // The algorithm requires two divisions by 2 and one by 3.
1724     // All divisions are known to be exact, that is, they do not produce
1725     // remainders, and all results are positive. The divisions by 2 are
1726     // implemented as right shifts which are relatively efficient, leaving
1727     // only an exact division by 3, which is done by a specialized
1728     // linear-time algorithm.
1729     t2 = v2.subtract(vm1).exactDivideBy3();

```

```

1730         tm1 = v1.subtract(vm1).shiftRight(1);
1731         t1 = v1.subtract(v0);
1732         t2 = t2.subtract(t1).shiftRight(1);
1733         t1 = t1.subtract(tm1).subtract(vinf);
1734         t2 = t2.subtract(vinf.shiftLeft(1));
1735         tm1 = tm1.subtract(t2);
1736
1737         // Number of bits to shift left.
1738         int ss = k*32;
1739
1740         BigInteger result = vinf.shiftLeft(ss).add(t2).shiftLeft(ss).add(t1).shiftLeft(ss).add(tm1)
1741             .shiftLeft(ss).add(v0);
1742
1743         if (a.signum != b.signum) {
1744             return result.negate();
1745         } else {
1746             return result;
1747         }
1748     }
1749
1750     /**
1751     * Returns a slice of a BigInteger for use in Toom-Cook multiplication.
1752     *
1753     * @param lowerSize The size of the lower-order bit slices.
1754     * @param upperSize The size of the higher-order bit slices.
1755     * @param slice The index of which slice is requested, which must be a
1756     * number from 0 to size-1. Slice 0 is the highest-order bits, and slice
1757     * size-1 are the lowest-order bits. Slice 0 may be of different size than
1758     * the other slices.
1759     * @param fullsize The size of the larger integer array, used to align
1760     * slices to the appropriate position when multiplying different-sized
1761     * numbers.
1762     */
1763     private BigInteger getToomSlice(int lowerSize, int upperSize, int slice,
1764                                     int fullsize) {
1765         int start, end, sliceSize, len, offset;
1766
1767         len = mag.length;
1768         offset = fullsize - len;
1769
1770         if (slice == 0) {
1771             start = 0 - offset;
1772             end = upperSize - 1 - offset;
1773         } else {
1774             start = upperSize + (slice-1)*lowerSize - offset;
1775             end = start + lowerSize - 1;
1776         }
1777
1778         if (start < 0) {
1779             start = 0;
1780         }
1781         if (end < 0) {

```

```

1782         return ZERO;
1783     }
1784
1785     sliceSize = (end-start) + 1;
1786
1787     if (sliceSize <= 0) {
1788         return ZERO;
1789     }
1790
1791     // While performing Toom-Cook, all slices are positive and
1792     // the sign is adjusted when the final number is composed.
1793     if (start == 0 && sliceSize >= len) {
1794         return this.abs();
1795     }
1796
1797     int intSlice[] = new int[sliceSize];
1798     System.arraycopy(mag, start, intSlice, 0, sliceSize);
1799
1800     return new BigInteger(trustedStripLeadingZeroInts(intSlice), 1);
1801 }
1802
1803 /**
1804  * Does an exact division (that is, the remainder is known to be zero)
1805  * of the specified number by 3. This is used in Toom-Cook
1806  * multiplication. This is an efficient algorithm that runs in linear
1807  * time. If the argument is not exactly divisible by 3, results are
1808  * undefined. Note that this is expected to be called with positive
1809  * arguments only.
1810  */
1811 private BigInteger exactDivideBy3() {
1812     int len = mag.length;
1813     int[] result = new int[len];
1814     long x, w, q, borrow;
1815     borrow = 0L;
1816     for (int i=len-1; i >= 0; i--) {
1817         x = (mag[i] & LONG_MASK);
1818         w = x - borrow;
1819         if (borrow > x) { // Did we make the number go negative?
1820             borrow = 1L;
1821         } else {
1822             borrow = 0L;
1823         }
1824
1825         // 0xAAAAAAAB is the modular inverse of 3 (mod 2^32). Thus,
1826         // the effect of this is to divide by 3 (mod 2^32).
1827         // This is much faster than division on most architectures.
1828         q = (w * 0xAAAAAAAB) & LONG_MASK;
1829         result[i] = (int) q;
1830
1831         // Now check the borrow. The second check can of course be
1832         // eliminated if the first fails.
1833         if (q >= 0x55555556L) {
1834             borrow++;

```

```

1835         if (q >= 0xAAAAAABL)
1836             borrow++;
1837     }
1838 }
1839 result = trustedStripLeadingZeroInts(result);
1840 return new BigInteger(result, signum);
1841 }
1842
1843 /**
1844  * Returns a new BigInteger representing n lower ints of the number.
1845  * This is used by Karatsuba multiplication and Karatsuba squaring.
1846  */
1847 private BigInteger getLower(int n) {
1848     int len = mag.length;
1849
1850     if (len <= n) {
1851         return abs();
1852     }
1853
1854     int lowerInts[] = new int[n];
1855     System.arraycopy(mag, len-n, lowerInts, 0, n);
1856
1857     return new BigInteger(trustedStripLeadingZeroInts(lowerInts), 1);
1858 }
1859
1860 /**
1861  * Returns a new BigInteger representing mag.length-n upper
1862  * ints of the number. This is used by Karatsuba multiplication and
1863  * Karatsuba squaring.
1864  */
1865 private BigInteger getUpper(int n) {
1866     int len = mag.length;
1867
1868     if (len <= n) {
1869         return ZERO;
1870     }
1871
1872     int upperLen = len - n;
1873     int upperInts[] = new int[upperLen];
1874     System.arraycopy(mag, 0, upperInts, 0, upperLen);
1875
1876     return new BigInteger(trustedStripLeadingZeroInts(upperInts), 1);
1877 }
1878
1879 // Squaring
1880
1881 /**
1882  * Returns a BigInteger whose value is  $(this^2)$ .
1883  *
1884  * @return  $this^2$ 
1885  */
1886 private BigInteger square() {
1887     if (signum == 0) {

```



```

1888         return ZERO;
1889     }
1890     int len = mag.length;
1891
1892     if (len < KARATSUBA_SQUARE_THRESHOLD) {
1893         int[] z = squareToLen(mag, len, null);
1894         return new BigInteger(trustedStripLeadingZeroInts(z), 1);
1895     } else {
1896         if (len < TOOM_COOK_SQUARE_THRESHOLD) {
1897             return squareKaratsuba();
1898         } else {
1899             return squareToomCook3();
1900         }
1901     }
1902 }
1903
1904 /**
1905  * Squares the contents of the int array x. The result is placed into the
1906  * int array z. The contents of x are not changed.
1907  */
1908 private static final int[] squareToLen(int[] x, int len, int[] z) {
1909     int zlen = len << 1;
1910     if (z == null || z.length < zlen)
1911         z = new int[zlen];
1912
1913     // Execute checks before calling intrinsified method.
1914     implSquareToLenChecks(x, len, z, zlen);
1915     return implSquareToLen(x, len, z, zlen);
1916 }
1917
1918 /**
1919  * Parameters validation.
1920  */
1921 private static void implSquareToLenChecks(int[] x, int len, int[] z, int zlen) throws
    RuntimeException {
1922     if (len < 1) {
1923         throw new IllegalArgumentException("invalid input length: " + len);
1924     }
1925     if (len > x.length) {
1926         throw new IllegalArgumentException("input length out of bound: " +
1927             len + " > " + x.length);
1928     }
1929     if (len * 2 > z.length) {
1930         throw new IllegalArgumentException("input length out of bound: " +
1931             (len * 2) + " > " + z.length);
1932     }
1933     if (zlen < 1) {
1934         throw new IllegalArgumentException("invalid input length: " + zlen);
1935     }
1936     if (zlen > z.length) {
1937         throw new IllegalArgumentException("input length out of bound: " +
1938             len + " > " + z.length);
1939     }

```

```

1940     }
1941
1942     /**
1943      * Java Runtime may use intrinsic for this method.
1944      */
1945     private static final int[] implSquareToLen(int[] x, int len, int[] z, int zlen) {
1946         /*
1947          * The algorithm used here is adapted from Colin Plumb's C library.
1948          * Technique: Consider the partial products in the multiplication
1949          * of "abcde" by itself:
1950          *
1951          *           a b c d e
1952          *         * a b c d e
1953          *       =====
1954          *           ae be ce de ee
1955          *         ad bd cd dd de
1956          *       ac bc cc cd ce
1957          *     ab bb bc bd be
1958          *   aa ab ac ad ae
1959          *
1960          * Note that everything above the main diagonal:
1961          *           ae be ce de = (abcd) * e
1962          *         ad bd cd      = (abc) * d
1963          *       ac bc          = (ab) * c
1964          *     ab              = (a) * b
1965          *
1966          * is a copy of everything below the main diagonal:
1967          *           de
1968          *         cd ce
1969          *       bc bd be
1970          *     ab ac ad ae
1971          *
1972          * Thus, the sum is 2 * (off the diagonal) + diagonal.
1973          *
1974          * This is accumulated beginning with the diagonal (which
1975          * consist of the squares of the digits of the input), which is then
1976          * divided by two, the off-diagonal added, and multiplied by two
1977          * again. The low bit is simply a copy of the low bit of the
1978          * input, so it doesn't need special care.
1979          */
1980
1981         // Store the squares, right shifted one bit (i.e., divided by 2)
1982         int lastProductLowWord = 0;
1983         for (int j=0, i=0; j < len; j++) {
1984             long piece = (x[j] & LONG_MASK);
1985             long product = piece * piece;
1986             z[i++] = (lastProductLowWord << 31) | (int)(product >>> 33);
1987             z[i++] = (int)(product >>> 1);
1988             lastProductLowWord = (int)product;
1989         }
1990
1991         // Add in off-diagonal sums
1992         for (int i=len, offset=1; i > 0; i--, offset+=2) {

```

```

1993         int t = x[i-1];
1994         t = mulAdd(z, x, offset, i-1, t);
1995         addOne(z, offset-1, i, t);
1996     }
1997
1998     // Shift back up and set low bit
1999     primitiveLeftShift(z, zlen, 1);
2000     z[zlen-1] |= x[len-1] & 1;
2001
2002     return z;
2003 }
2004
2005 /**
2006  * Squares a BigInteger using the Karatsuba squaring algorithm. It should
2007  * be used when both numbers are larger than a certain threshold (found
2008  * experimentally). It is a recursive divide-and-conquer algorithm that
2009  * has better asymptotic performance than the algorithm used in
2010  * squareToLen.
2011  */
2012 private BigInteger squareKaratsuba() {
2013     int half = (mag.length+1) / 2;
2014
2015     BigInteger xl = getLower(half);
2016     BigInteger xh = getUpper(half);
2017
2018     BigInteger xhs = xh.square(); // xhs = xh^2
2019     BigInteger xls = xl.square(); // xls = xl^2
2020
2021     // xh^2 << 64 + ((xl+xh)^2 - (xh^2 + xl^2)) << 32 + xl^2
2022     return xhs.shiftLeft(half*32).add(xl.add(xh).square().subtract(xhs.add(xls))).shiftLeft(
2023         half*32).add(xls);
2024 }
2025
2026 /**
2027  * Squares a BigInteger using the 3-way Toom-Cook squaring algorithm. It
2028  * should be used when both numbers are larger than a certain threshold
2029  * (found experimentally). It is a recursive divide-and-conquer algorithm
2030  * that has better asymptotic performance than the algorithm used in
2031  * squareToLen or squareKaratsuba.
2032  */
2033 private BigInteger squareToomCook3() {
2034     int len = mag.length;
2035
2036     // k is the size (in ints) of the lower-order slices.
2037     int k = (len+2)/3; // Equal to ceil(largest/3)
2038
2039     // r is the size (in ints) of the highest-order slice.
2040     int r = len - 2*k;
2041
2042     // Obtain slices of the numbers. a2 is the most significant
2043     // bits of the number, and a0 the least significant.
2044     BigInteger a0, a1, a2;
2045     a2 = getToomSlice(k, r, 0, len);

```

```

2045     a1 = getToomSlice(k, r, 1, len);
2046     a0 = getToomSlice(k, r, 2, len);
2047     BigInteger v0, v1, v2, vm1, vinf, t1, t2, tm1, da1;
2048
2049     v0 = a0.square();
2050     da1 = a2.add(a0);
2051     vm1 = da1.subtract(a1).square();
2052     da1 = da1.add(a1);
2053     v1 = da1.square();
2054     vinf = a2.square();
2055     v2 = da1.add(a2).shiftLeft(1).subtract(a0).square();
2056
2057     // The algorithm requires two divisions by 2 and one by 3.
2058     // All divisions are known to be exact, that is, they do not produce
2059     // remainders, and all results are positive. The divisions by 2 are
2060     // implemented as right shifts which are relatively efficient, leaving
2061     // only a division by 3.
2062     // The division by 3 is done by an optimized algorithm for this case.
2063     t2 = v2.subtract(vm1).exactDivideBy3();
2064     tm1 = v1.subtract(vm1).shiftRight(1);
2065     t1 = v1.subtract(v0);
2066     t2 = t2.subtract(t1).shiftRight(1);
2067     t1 = t1.subtract(tm1).subtract(vinf);
2068     t2 = t2.subtract(vinf.shiftLeft(1));
2069     tm1 = tm1.subtract(t2);
2070
2071     // Number of bits to shift left.
2072     int ss = k*32;
2073
2074     return vinf.shiftLeft(ss).add(t2).shiftLeft(ss).add(t1).shiftLeft(ss).add(tm1).shiftLeft(ss)
2075         .add(v0);
2076
2077     // Division
2078
2079     /**
2080     * Returns a BigInteger whose value is {@code (this / val)}.
2081     *
2082     * @param val value by which this BigInteger is to be divided.
2083     * @return {@code this / val}
2084     * @throws ArithmeticException if {@code val} is zero.
2085     */
2086     public BigInteger divide(BigInteger val) {
2087         if (val.mag.length < BURNIKEL_ZIEGLER_THRESHOLD ||
2088             mag.length - val.mag.length < BURNIKEL_ZIEGLER_OFFSET) {
2089             return divideKnuth(val);
2090         } else {
2091             return divideBurnikelZiegler(val);
2092         }
2093     }
2094
2095     /**
2096     * Returns a BigInteger whose value is {@code (this / val)} using an  $O(n^2)$  algorithm from

```

```

    Knuth.
2097     *
2098     * @param val value by which this BigInteger is to be divided.
2099     * @return {@code this / val}
2100     * @throws ArithmeticException if {@code val} is zero.
2101     * @see MutableBigInteger#divideKnuth(MutableBigInteger, MutableBigInteger, boolean)
2102     */
2103     private BigInteger divideKnuth(BigInteger val) {
2104         MutableBigInteger q = new MutableBigInteger(),
2105             a = new MutableBigInteger(this.mag),
2106             b = new MutableBigInteger(val.mag);
2107
2108         a.divideKnuth(b, q, false);
2109         return q.toBigInteger(this.signum * val.signum);
2110     }
2111
2112     /**
2113     * Returns an array of two BigIntegers containing {@code (this / val)}
2114     * followed by {@code (this % val)}.
2115     *
2116     * @param val value by which this BigInteger is to be divided, and the
2117     *             remainder computed.
2118     * @return an array of two BigIntegers: the quotient {@code (this / val)}
2119     *         is the initial element, and the remainder {@code (this % val)}
2120     *         is the final element.
2121     * @throws ArithmeticException if {@code val} is zero.
2122     */
2123     public BigInteger[] divideAndRemainder(BigInteger val) {
2124         if (val.mag.length < BURNIKEL_ZIEGLER_THRESHOLD ||
2125             mag.length - val.mag.length < BURNIKEL_ZIEGLER_OFFSET) {
2126             return divideAndRemainderKnuth(val);
2127         } else {
2128             return divideAndRemainderBurnikelZiegler(val);
2129         }
2130     }
2131
2132     /** Long division */
2133     private BigInteger[] divideAndRemainderKnuth(BigInteger val) {
2134         BigInteger[] result = new BigInteger[2];
2135         MutableBigInteger q = new MutableBigInteger(),
2136             a = new MutableBigInteger(this.mag),
2137             b = new MutableBigInteger(val.mag);
2138         MutableBigInteger r = a.divideKnuth(b, q);
2139         result[0] = q.toBigInteger(this.signum == val.signum ? 1 : -1);
2140         result[1] = r.toBigInteger(this.signum);
2141         return result;
2142     }
2143
2144     /**
2145     * Returns a BigInteger whose value is {@code (this % val)}.
2146     *
2147     * @param val value by which this BigInteger is to be divided, and the
2148     *             remainder computed.

```

```

2149     * @return {@code this % val}
2150     * @throws ArithmeticException if {@code val} is zero.
2151     */
2152     public BigInteger remainder(BigInteger val) {
2153         if (val.mag.length < BURNIKEL_ZIEGLER_THRESHOLD ||
2154             mag.length - val.mag.length < BURNIKEL_ZIEGLER_OFFSET) {
2155             return remainderKnuth(val);
2156         } else {
2157             return remainderBurnikelZiegler(val);
2158         }
2159     }
2160
2161     /** Long division */
2162     private BigInteger remainderKnuth(BigInteger val) {
2163         MutableBigInteger q = new MutableBigInteger(),
2164             a = new MutableBigInteger(this.mag),
2165             b = new MutableBigInteger(val.mag);
2166
2167         return a.divideKnuth(b, q).toBigInteger(this.signum);
2168     }
2169
2170     /**
2171      * Calculates {@code this / val} using the Burnikel-Ziegler algorithm.
2172      * @param val the divisor
2173      * @return {@code this / val}
2174      */
2175     private BigInteger divideBurnikelZiegler(BigInteger val) {
2176         return divideAndRemainderBurnikelZiegler(val)[0];
2177     }
2178
2179     /**
2180      * Calculates {@code this % val} using the Burnikel-Ziegler algorithm.
2181      * @param val the divisor
2182      * @return {@code this % val}
2183      */
2184     private BigInteger remainderBurnikelZiegler(BigInteger val) {
2185         return divideAndRemainderBurnikelZiegler(val)[1];
2186     }
2187
2188     /**
2189      * Computes {@code this / val} and {@code this % val} using the
2190      * Burnikel-Ziegler algorithm.
2191      * @param val the divisor
2192      * @return an array containing the quotient and remainder
2193      */
2194     private BigInteger[] divideAndRemainderBurnikelZiegler(BigInteger val) {
2195         MutableBigInteger q = new MutableBigInteger();
2196         MutableBigInteger r = new MutableBigInteger(this).divideAndRemainderBurnikelZiegler(new
            MutableBigInteger(val), q);
2197         BigInteger qBigInt = q.isZero() ? ZERO : q.toBigInteger(signum*val.signum);
2198         BigInteger rBigInt = r.isZero() ? ZERO : r.toBigInteger(signum);
2199         return new BigInteger[] {qBigInt, rBigInt};
2200     }

```

```

2201
2202 /**
2203  * Returns a BigInteger whose value is <tt>(this<sup>exponent</sup></tt>.
2204  * Note that {@code exponent} is an integer rather than a BigInteger.
2205  *
2206  * @param exponent exponent to which this BigInteger is to be raised.
2207  * @return <tt>this<sup>exponent</sup></tt>
2208  * @throws ArithmeticException {@code exponent} is negative. (This would
2209  *         cause the operation to yield a non-integer value.)
2210  */
2211 public BigInteger pow(int exponent) {
2212     if (exponent < 0) {
2213         throw new ArithmeticException("Negative exponent");
2214     }
2215     if (signum == 0) {
2216         return (exponent == 0 ? ONE : this);
2217     }
2218
2219     BigInteger partToSquare = this.abs();
2220
2221     // Factor out powers of two from the base, as the exponentiation of
2222     // these can be done by left shifts only.
2223     // The remaining part can then be exponentiated faster. The
2224     // powers of two will be multiplied back at the end.
2225     int powersOfTwo = partToSquare.getLowestSetBit();
2226     long bitsToShift = (long)powersOfTwo * exponent;
2227     if (bitsToShift > Integer.MAX_VALUE) {
2228         reportOverflow();
2229     }
2230
2231     int remainingBits;
2232
2233     // Factor the powers of two out quickly by shifting right, if needed.
2234     if (powersOfTwo > 0) {
2235         partToSquare = partToSquare.shiftRight(powersOfTwo);
2236         remainingBits = partToSquare.bitLength();
2237         if (remainingBits == 1) { // Nothing left but +/- 1?
2238             if (signum < 0 && (exponent&1) == 1) {
2239                 return NEGATIVE_ONE.shiftLeft(powersOfTwo*exponent);
2240             } else {
2241                 return ONE.shiftLeft(powersOfTwo*exponent);
2242             }
2243         }
2244     } else {
2245         remainingBits = partToSquare.bitLength();
2246         if (remainingBits == 1) { // Nothing left but +/- 1?
2247             if (signum < 0 && (exponent&1) == 1) {
2248                 return NEGATIVE_ONE;
2249             } else {
2250                 return ONE;
2251             }
2252         }
2253     }

```

```

2254
2255     // This is a quick way to approximate the size of the result,
2256     // similar to doing log2[n] * exponent. This will give an upper bound
2257     // of how big the result can be, and which algorithm to use.
2258     long scaleFactor = (long)remainingBits * exponent;
2259
2260     // Use slightly different algorithms for small and large operands.
2261     // See if the result will safely fit into a long. (Largest 2^63-1)
2262     if (partToSquare.mag.length == 1 && scaleFactor <= 62) {
2263         // Small number algorithm. Everything fits into a long.
2264         int newSign = (signum < 0 && (exponent & 1) == 1 ? -1 : 1);
2265         long result = 1;
2266         long baseToPow2 = partToSquare.mag[0] & LONG_MASK;
2267
2268         int workingExponent = exponent;
2269
2270         // Perform exponentiation using repeated squaring trick
2271         while (workingExponent != 0) {
2272             if ((workingExponent & 1) == 1) {
2273                 result = result * baseToPow2;
2274             }
2275
2276             if ((workingExponent >>= 1) != 0) {
2277                 baseToPow2 = baseToPow2 * baseToPow2;
2278             }
2279         }
2280
2281         // Multiply back the powers of two (quickly, by shifting left)
2282         if (powersOfTwo > 0) {
2283             if (bitsToShift + scaleFactor <= 62) { // Fits in long?
2284                 return valueOf((result << bitsToShift) * newSign);
2285             } else {
2286                 return valueOf(result * newSign).shiftLeft((int) bitsToShift);
2287             }
2288         }
2289         else {
2290             return valueOf(result * newSign);
2291         }
2292     } else {
2293         // Large number algorithm. This is basically identical to
2294         // the algorithm above, but calls multiply() and square()
2295         // which may use more efficient algorithms for large numbers.
2296         BigInteger answer = ONE;
2297
2298         int workingExponent = exponent;
2299         // Perform exponentiation using repeated squaring trick
2300         while (workingExponent != 0) {
2301             if ((workingExponent & 1) == 1) {
2302                 answer = answer.multiply(partToSquare);
2303             }
2304
2305             if ((workingExponent >>= 1) != 0) {
2306                 partToSquare = partToSquare.square();

```



```

2307     }
2308 }
2309 // Multiply back the (exponentiated) powers of two (quickly,
2310 // by shifting left)
2311 if (powersOfTwo > 0) {
2312     answer = answer.shiftLeft(powersOfTwo*exponent);
2313 }
2314
2315 if (signum < 0 && (exponent&1) == 1) {
2316     return answer.negate();
2317 } else {
2318     return answer;
2319 }
2320 }
2321 }
2322
2323 /**
2324  * Returns a BigInteger whose value is the greatest common divisor of
2325  * {@code abs(this)} and {@code abs(val)}. Returns 0 if
2326  * {@code this == 0 && val == 0}.
2327  *
2328  * @param val value with which the GCD is to be computed.
2329  * @return {@code GCD(abs(this), abs(val))}
2330  */
2331 public BigInteger gcd(BigInteger val) {
2332     if (val.signum == 0)
2333         return this.abs();
2334     else if (this.signum == 0)
2335         return val.abs();
2336
2337     MutableBigInteger a = new MutableBigInteger(this);
2338     MutableBigInteger b = new MutableBigInteger(val);
2339
2340     MutableBigInteger result = a.hybridGCD(b);
2341
2342     return result.toBigInteger(1);
2343 }
2344
2345 /**
2346  * Package private method to return bit length for an integer.
2347  */
2348 static int bitLengthForInt(int n) {
2349     return 32 - Integer.numberOfLeadingZeros(n);
2350 }
2351
2352 /**
2353  * Left shift int array a up to len by n bits. Returns the array that
2354  * results from the shift since space may have to be reallocated.
2355  */
2356 private static int[] leftShift(int[] a, int len, int n) {
2357     int nInts = n >>> 5;
2358     int nBits = n&0x1F;
2359     int bitsInHighWord = bitLengthForInt(a[0]);

```

```

2360
2361 // If shift can be done without recopy, do so
2362 if (n <= (32-bitsInHighWord)) {
2363     primitiveLeftShift(a, len, nBits);
2364     return a;
2365 } else { // Array must be resized
2366     if (nBits <= (32-bitsInHighWord)) {
2367         int result[] = new int[nInts+len];
2368         System.arraycopy(a, 0, result, 0, len);
2369         primitiveLeftShift(result, result.length, nBits);
2370         return result;
2371     } else {
2372         int result[] = new int[nInts+len+1];
2373         System.arraycopy(a, 0, result, 0, len);
2374         primitiveRightShift(result, result.length, 32 - nBits);
2375         return result;
2376     }
2377 }
2378 }
2379
2380 // shifts a up to len right n bits assumes no leading zeros, 0<n<32
2381 static void primitiveRightShift(int[] a, int len, int n) {
2382     int n2 = 32 - n;
2383     for (int i=len-1, c=a[i]; i > 0; i--) {
2384         int b = c;
2385         c = a[i-1];
2386         a[i] = (c << n2) | (b >>> n);
2387     }
2388     a[0] >>>= n;
2389 }
2390
2391 // shifts a up to len left n bits assumes no leading zeros, 0<=n<32
2392 static void primitiveLeftShift(int[] a, int len, int n) {
2393     if (len == 0 || n == 0)
2394         return;
2395
2396     int n2 = 32 - n;
2397     for (int i=0, c=a[i], m=i+len-1; i < m; i++) {
2398         int b = c;
2399         c = a[i+1];
2400         a[i] = (b << n) | (c >>> n2);
2401     }
2402     a[len-1] <<= n;
2403 }
2404
2405 /**
2406  * Calculate bitlength of contents of the first len elements an int array,
2407  * assuming there are no leading zero ints.
2408  */
2409 private static int bitLength(int[] val, int len) {
2410     if (len == 0)
2411         return 0;
2412     return ((len - 1) << 5) + bitLengthForInt(val[0]);

```

```

2413     }
2414
2415     /**
2416      * Returns a BigInteger whose value is the absolute value of this
2417      * BigInteger.
2418      *
2419      * @return {@code abs(this)}
2420      */
2421     public BigInteger abs() {
2422         return (signum >= 0 ? this : this.negate());
2423     }
2424
2425     /**
2426      * Returns a BigInteger whose value is {@code (-this)}.
2427      *
2428      * @return {@code -this}
2429      */
2430     public BigInteger negate() {
2431         return new BigInteger(this.mag, -this.signum);
2432     }
2433
2434     /**
2435      * Returns the signum function of this BigInteger.
2436      *
2437      * @return -1, 0 or 1 as the value of this BigInteger is negative, zero or
2438      *         positive.
2439      */
2440     public int signum() {
2441         return this.signum;
2442     }
2443
2444     // Modular Arithmetic Operations
2445
2446     /**
2447      * Returns a BigInteger whose value is {@code (this mod m)}. This method
2448      * differs from {@code remainder} in that it always returns a
2449      * <i>non-negative</i> BigInteger.
2450      *
2451      * @param m the modulus.
2452      * @return {@code this mod m}
2453      * @throws ArithmeticException {@code m} <= 0
2454      * @see #remainder
2455      */
2456     public BigInteger mod(BigInteger m) {
2457         if (m.signum <= 0)
2458             throw new ArithmeticException("BigInteger: modulus not positive");
2459
2460         BigInteger result = this.remainder(m);
2461         return (result.signum >= 0 ? result : result.add(m));
2462     }
2463
2464     /**
2465      * Returns a BigInteger whose value is

```

```

2466 * <tt>(this<sup>exponent</sup> mod m)</tt>. (Unlike {@code pow}, this
2467 * method permits negative exponents.)
2468 *
2469 * @param exponent the exponent.
2470 * @param m the modulus.
2471 * @return <tt>this<sup>exponent</sup> mod m</tt>
2472 * @throws ArithmeticException {@code m} ≤ 0 or the exponent is
2473 *         negative and this BigInteger is not <i>relatively
2474 *         prime</i> to {@code m}.
2475 * @see    #modInverse
2476 */
2477 public BigInteger modPow(BigInteger exponent, BigInteger m) {
2478     if (m.signum <= 0)
2479         throw new ArithmeticException("BigInteger: modulus not positive");
2480
2481     // Trivial cases
2482     if (exponent.signum == 0)
2483         return (m.equals(ONE) ? ZERO : ONE);
2484
2485     if (this.equals(ONE))
2486         return (m.equals(ONE) ? ZERO : ONE);
2487
2488     if (this.equals(ZERO) && exponent.signum >= 0)
2489         return ZERO;
2490
2491     if (this.equals(negConst[1]) && (!exponent.testBit(0)))
2492         return (m.equals(ONE) ? ZERO : ONE);
2493
2494     boolean invertResult;
2495     if ((invertResult = (exponent.signum < 0)))
2496         exponent = exponent.negate();
2497
2498     BigInteger base = (this.signum < 0 || this.compareTo(m) >= 0
2499         ? this.mod(m) : this);
2500     BigInteger result;
2501     if (m.testBit(0)) { // odd modulus
2502         result = base.oddModPow(exponent, m);
2503     } else {
2504         /*
2505          * Even modulus. Tear it into an "odd part" (m1) and power of two
2506          * (m2), exponentiate mod m1, manually exponentiate mod m2, and
2507          * use Chinese Remainder Theorem to combine results.
2508          */
2509
2510         // Tear m apart into odd part (m1) and power of 2 (m2)
2511         int p = m.getLowestSetBit(); // Max pow of 2 that divides m
2512
2513         BigInteger m1 = m.shiftRight(p); // m/2**p
2514         BigInteger m2 = ONE.shiftLeft(p); // 2**p
2515
2516         // Calculate new base from m1
2517         BigInteger base2 = (this.signum < 0 || this.compareTo(m1) >= 0
2518             ? this.mod(m1) : this);

```

```

2519
2520     // Caculate (base ** exponent) mod m1.
2521     BigInteger a1 = (m1.equals(ONE) ? ZERO :
2522         base2.oddModPow(exponent, m1));
2523
2524     // Calculate (this ** exponent) mod m2
2525     BigInteger a2 = base.modPow2(exponent, p);
2526
2527     // Combine results using Chinese Remainder Theorem
2528     BigInteger y1 = m2.modInverse(m1);
2529     BigInteger y2 = m1.modInverse(m2);
2530
2531     if (m.mag.length < MAX_MAG_LENGTH / 2) {
2532         result = a1.multiply(m2).multiply(y1).add(a2.multiply(m1).multiply(y2)).mod(m);
2533     } else {
2534         MutableBigInteger t1 = new MutableBigInteger();
2535         new MutableBigInteger(a1.multiply(m2)).multiply(new MutableBigInteger(y1), t1);
2536         MutableBigInteger t2 = new MutableBigInteger();
2537         new MutableBigInteger(a2.multiply(m1)).multiply(new MutableBigInteger(y2), t2);
2538         t1.add(t2);
2539         MutableBigInteger q = new MutableBigInteger();
2540         result = t1.divide(new MutableBigInteger(m), q).toBigInteger();
2541     }
2542 }
2543
2544     return (invertResult ? result.modInverse(m) : result);
2545 }
2546
2547 // Montgomery multiplication. These are wrappers for
2548 // implMontgomeryXX routines which are expected to be replaced by
2549 // virtual machine intrinsics. We don't use the intrinsics for
2550 // very large operands: MONTGOMERY_INTRINSIC_THRESHOLD should be
2551 // larger than any reasonable crypto key.
2552 private static int[] montgomeryMultiply(int[] a, int[] b, int[] n, int len, long inv,
2553     int[] product) {
2554     implMontgomeryMultiplyChecks(a, b, n, len, product);
2555     if (len > MONTGOMERY_INTRINSIC_THRESHOLD) {
2556         // Very long argument: do not use an intrinsic
2557         product = multiplyToLen(a, len, b, len, product);
2558         return montReduce(product, n, len, (int)inv);
2559     } else {
2560         return implMontgomeryMultiply(a, b, n, len, inv, materialize(product, len));
2561     }
2562 }
2563 private static int[] montgomerySquare(int[] a, int[] n, int len, long inv,
2564     int[] product) {
2565     implMontgomeryMultiplyChecks(a, a, n, len, product);
2566     if (len > MONTGOMERY_INTRINSIC_THRESHOLD) {
2567         // Very long argument: do not use an intrinsic
2568         product = squareToLen(a, len, product);
2569         return montReduce(product, n, len, (int)inv);
2570     } else {
2571         return implMontgomerySquare(a, n, len, inv, materialize(product, len));

```

```

2572     }
2573 }
2574
2575 // Range-check everything.
2576 private static void implMontgomeryMultiplyChecks
2577     (int[] a, int[] b, int[] n, int len, int[] product) throws RuntimeException {
2578     if (len % 2 != 0) {
2579         throw new IllegalArgumentException("input array length must be even: " + len);
2580     }
2581
2582     if (len < 1) {
2583         throw new IllegalArgumentException("invalid input length: " + len);
2584     }
2585
2586     if (len > a.length ||
2587         len > b.length ||
2588         len > n.length ||
2589         (product != null && len > product.length)) {
2590         throw new IllegalArgumentException("input array length out of bound: " + len);
2591     }
2592 }
2593
2594 // Make sure that the int array z (which is expected to contain
2595 // the result of a Montgomery multiplication) is present and
2596 // sufficiently large.
2597 private static int[] materialize(int[] z, int len) {
2598     if (z == null || z.length < len)
2599         z = new int[len];
2600     return z;
2601 }
2602
2603 // These methods are intended to be replaced by virtual machine
2604 // intrinsics.
2605 private static int[] implMontgomeryMultiply(int[] a, int[] b, int[] n, int len,
2606                                             long inv, int[] product) {
2607     product = multiplyToLen(a, len, b, len, product);
2608     return montReduce(product, n, len, (int)inv);
2609 }
2610 private static int[] implMontgomerySquare(int[] a, int[] n, int len,
2611                                           long inv, int[] product) {
2612     product = squareToLen(a, len, product);
2613     return montReduce(product, n, len, (int)inv);
2614 }
2615
2616 static int[] bnExpModThreshTable = {7, 25, 81, 241, 673, 1793,
2617                                     Integer.MAX_VALUE}; // Sentinel
2618
2619 /**
2620  * Returns a BigInteger whose value is x to the power of y mod z.
2621  * Assumes: z is odd && x < z.
2622  */
2623 private BigInteger oddModPow(BigInteger y, BigInteger z) {
2624     /*

```

```

2625 * The algorithm is adapted from Colin Plumb's C library.
2626 *
2627 * The window algorithm:
2628 * The idea is to keep a running product of  $b1 = n^{(\text{high-order bits of exp})}$ 
2629 * and then keep appending exponent bits to it. The following patterns
2630 * apply to a 3-bit window ( $k = 3$ ):
2631 * To append 0: square
2632 * To append 1: square, multiply by  $n^1$ 
2633 * To append 10: square, multiply by  $n^1$ , square
2634 * To append 11: square, square, multiply by  $n^3$ 
2635 * To append 100: square, multiply by  $n^1$ , square, square
2636 * To append 101: square, square, square, multiply by  $n^5$ 
2637 * To append 110: square, square, multiply by  $n^3$ , square
2638 * To append 111: square, square, square, multiply by  $n^7$ 
2639 *
2640 * Since each pattern involves only one multiply, the longer the pattern
2641 * the better, except that a 0 (no multiplies) can be appended directly.
2642 * We precompute a table of odd powers of  $n$ , up to  $2^k$ , and can then
2643 * multiply  $k$  bits of exponent at a time. Actually, assuming random
2644 * exponents, there is on average one zero bit between needs to
2645 * multiply ( $1/2$  of the time there's none,  $1/4$  of the time there's 1,
2646 *  $1/8$  of the time, there's 2,  $1/32$  of the time, there's 3, etc.), so
2647 * you have to do one multiply per  $k+1$  bits of exponent.
2648 *
2649 * The loop walks down the exponent, squaring the result buffer as
2650 * it goes. There is a  $wbits+1$  bit lookahead buffer, buf, that is
2651 * filled with the upcoming exponent bits. (What is read after the
2652 * end of the exponent is unimportant, but it is filled with zero here.)
2653 * When the most-significant bit of this buffer becomes set, i.e.
2654 * (buf & tblmask) != 0, we have to decide what pattern to multiply
2655 * by, and when to do it. We decide, remember to do it in future
2656 * after a suitable number of squarings have passed (e.g. a pattern
2657 * of "100" in the buffer requires that we multiply by  $n^1$  immediately;
2658 * a pattern of "110" calls for multiplying by  $n^3$  after one more
2659 * squaring), clear the buffer, and continue.
2660 *
2661 * When we start, there is one more optimization: the result buffer
2662 * is implicitly one, so squaring it or multiplying by it can be
2663 * optimized away. Further, if we start with a pattern like "100"
2664 * in the lookahead window, rather than placing  $n$  into the buffer
2665 * and then starting to square it, we have already computed  $n^2$ 
2666 * to compute the odd-powers table, so we can place that into
2667 * the buffer and save a squaring.
2668 *
2669 * This means that if you have a  $k$ -bit window, to compute  $n^z$ ,
2670 * where  $z$  is the high  $k$  bits of the exponent,  $1/2$  of the time
2671 * it requires no squarings.  $1/4$  of the time, it requires 1
2672 * squaring, ...  $1/2^{(k-1)}$  of the time, it requires  $k-2$  squarings.
2673 * And the remaining  $1/2^{(k-1)}$  of the time, the top  $k$  bits are a
2674 * 1 followed by  $k-1$  0 bits, so it again only requires  $k-2$ 
2675 * squarings, not  $k-1$ . The average of these is 1. Add that
2676 * to the one squaring we have to do to compute the table,
2677 * and you'll see that a  $k$ -bit window saves  $k-2$  squarings

```

```

2678     * as well as reducing the multiplies. (It actually doesn't
2679     * hurt in the case k = 1, either.)
2680     */
2681     // Special case for exponent of one
2682     if (y.equals(ONE))
2683         return this;
2684
2685     // Special case for base of zero
2686     if (signum == 0)
2687         return ZERO;
2688
2689     int[] base = mag.clone();
2690     int[] exp = y.mag;
2691     int[] mod = z.mag;
2692     int modLen = mod.length;
2693
2694     // Make modLen even. It is conventional to use a cryptographic
2695     // modulus that is 512, 768, 1024, or 2048 bits, so this code
2696     // will not normally be executed. However, it is necessary for
2697     // the correct functioning of the HotSpot intrinsics.
2698     if ((modLen & 1) != 0) {
2699         int[] x = new int[modLen + 1];
2700         System.arraycopy(mod, 0, x, 1, modLen);
2701         mod = x;
2702         modLen++;
2703     }
2704
2705     // Select an appropriate window size
2706     int wbits = 0;
2707     int ebits = bitLength(exp, exp.length);
2708     // if exponent is 65537 (0x10001), use minimum window size
2709     if ((ebits != 17) || (exp[0] != 65537)) {
2710         while (ebits > bnExpModThreshTable[wbits]) {
2711             wbits++;
2712         }
2713     }
2714
2715     // Calculate appropriate table size
2716     int tblmask = 1 << wbits;
2717
2718     // Allocate table for precomputed odd powers of base in Montgomery form
2719     int[][] table = new int[tblmask][];
2720     for (int i=0; i < tblmask; i++)
2721         table[i] = new int[modLen];
2722
2723     // Compute the modular inverse of the least significant 64-bit
2724     // digit of the modulus
2725     long n0 = (mod[modLen-1] & LONG_MASK) + ((mod[modLen-2] & LONG_MASK) << 32);
2726     long inv = -MutableBigInteger.inverseMod64(n0);
2727
2728     // Convert base to Montgomery form
2729     int[] a = leftShift(base, base.length, modLen << 5);
2730

```



```

2731     MutableBigInteger q = new MutableBigInteger(),
2732                 a2 = new MutableBigInteger(a),
2733                 b2 = new MutableBigInteger(mod);
2734     b2.normalize(); // MutableBigInteger.divide() assumes that its
2735                     // divisor is in normal form.
2736
2737     MutableBigInteger r= a2.divide(b2, q);
2738     table[0] = r.toIntArray();
2739
2740     // Pad table[0] with leading zeros so its length is at least modLen
2741     if (table[0].length < modLen) {
2742         int offset = modLen - table[0].length;
2743         int[] t2 = new int[modLen];
2744         System.arraycopy(table[0], 0, t2, offset, table[0].length);
2745         table[0] = t2;
2746     }
2747
2748     // Set b to the square of the base
2749     int[] b = montgomerySquare(table[0], mod, modLen, inv, null);
2750
2751     // Set t to high half of b
2752     int[] t = Arrays.copyOf(b, modLen);
2753
2754     // Fill in the table with odd powers of the base
2755     for (int i=1; i < tblmask; i++) {
2756         table[i] = montgomeryMultiply(t, table[i-1], mod, modLen, inv, null);
2757     }
2758
2759     // Pre load the window that slides over the exponent
2760     int bitpos = 1 << ((ebits-1) & (32-1));
2761
2762     int buf = 0;
2763     int elen = exp.length;
2764     int eIndex = 0;
2765     for (int i = 0; i <= wbits; i++) {
2766         buf = (buf << 1) | (((exp[eIndex] & bitpos) != 0)?1:0);
2767         bitpos >>= 1;
2768         if (bitpos == 0) {
2769             eIndex++;
2770             bitpos = 1 << (32-1);
2771             elen--;
2772         }
2773     }
2774
2775     int multpos = ebits;
2776
2777     // The first iteration, which is hoisted out of the main loop
2778     ebits--;
2779     boolean isone = true;
2780
2781     multpos = ebits - wbits;
2782     while ((buf & 1) == 0) {
2783         buf >>= 1;

```

```

2784     multpos++;
2785 }
2786
2787 int[] mult = table[buf >>> 1];
2788
2789 buf = 0;
2790 if (multpos == ebits)
2791     isone = false;
2792
2793 // The main loop
2794 while (true) {
2795     ebits--;
2796     // Advance the window
2797     buf <<= 1;
2798
2799     if (elen != 0) {
2800         buf |= ((exp[eIndex] & bitpos) != 0) ? 1 : 0;
2801         bitpos >>= 1;
2802         if (bitpos == 0) {
2803             eIndex++;
2804             bitpos = 1 << (32-1);
2805             elen--;
2806         }
2807     }
2808
2809     // Examine the window for pending multiplies
2810     if ((buf & tblmask) != 0) {
2811         multpos = ebits - wbits;
2812         while ((buf & 1) == 0) {
2813             buf >>= 1;
2814             multpos++;
2815         }
2816         mult = table[buf >>> 1];
2817         buf = 0;
2818     }
2819
2820     // Perform multiply
2821     if (ebits == multpos) {
2822         if (isone) {
2823             b = mult.clone();
2824             isone = false;
2825         } else {
2826             t = b;
2827             a = montgomeryMultiply(t, mult, mod, modLen, inv, a);
2828             t = a; a = b; b = t;
2829         }
2830     }
2831
2832     // Check if done
2833     if (ebits == 0)
2834         break;
2835
2836     // Square the input

```

```

2837         if (!isone) {
2838             t = b;
2839             a = montgomerySquare(t, mod, modLen, inv, a);
2840             t = a; a = b; b = t;
2841         }
2842     }
2843
2844     // Convert result out of Montgomery form and return
2845     int[] t2 = new int[2*modLen];
2846     System.arraycopy(b, 0, t2, modLen, modLen);
2847
2848     b = montReduce(t2, mod, modLen, (int)inv);
2849
2850     t2 = Arrays.copyOf(b, modLen);
2851
2852     return new BigInteger(1, t2);
2853 }
2854
2855 /**
2856  * Montgomery reduce n, modulo mod. This reduces modulo mod and divides
2857  * by 2^(32*mmlen). Adapted from Colin Plumb's C library.
2858  */
2859 private static int[] montReduce(int[] n, int[] mod, int mlen, int inv) {
2860     int c=0;
2861     int len = mlen;
2862     int offset=0;
2863
2864     do {
2865         int nEnd = n[n.length-1-offset];
2866         int carry = mulAdd(n, mod, offset, mlen, inv * nEnd);
2867         c += addOne(n, offset, mlen, carry);
2868         offset++;
2869     } while (--len > 0);
2870
2871     while (c > 0)
2872         c += subN(n, mod, mlen);
2873
2874     while (intArrayCmpToLen(n, mod, mlen) >= 0)
2875         subN(n, mod, mlen);
2876
2877     return n;
2878 }
2879
2880
2881 /**
2882  * Returns -1, 0 or +1 as big-endian unsigned int array arg1 is less than,
2883  * equal to, or greater than arg2 up to length len.
2884  */
2885 private static int intArrayCmpToLen(int[] arg1, int[] arg2, int len) {
2886     for (int i=0; i < len; i++) {
2887         long b1 = arg1[i] & LONG_MASK;
2888         long b2 = arg2[i] & LONG_MASK;
2889         if (b1 < b2)

```

```

2890         return -1;
2891     if (b1 > b2)
2892         return 1;
2893     }
2894     return 0;
2895 }
2896
2897 /**
2898  * Subtracts two numbers of same length, returning borrow.
2899  */
2900 private static int subN(int[] a, int[] b, int len) {
2901     long sum = 0;
2902
2903     while (--len >= 0) {
2904         sum = (a[len] & LONG_MASK) -
2905             (b[len] & LONG_MASK) + (sum >> 32);
2906         a[len] = (int)sum;
2907     }
2908
2909     return (int)(sum >> 32);
2910 }
2911
2912 /**
2913  * Multiply an array by one word k and add to result, return the carry
2914  */
2915 static int mulAdd(int[] out, int[] in, int offset, int len, int k) {
2916     implMulAddCheck(out, in, offset, len, k);
2917     return implMulAdd(out, in, offset, len, k);
2918 }
2919
2920 /**
2921  * Parameters validation.
2922  */
2923 private static void implMulAddCheck(int[] out, int[] in, int offset, int len, int k) {
2924     if (len > in.length) {
2925         throw new IllegalArgumentException("input length is out of bound: " + len + " > " + in.
2926             length);
2927     }
2928     if (offset < 0) {
2929         throw new IllegalArgumentException("input offset is invalid: " + offset);
2930     }
2931     if (offset > (out.length - 1)) {
2932         throw new IllegalArgumentException("input offset is out of bound: " + offset + " > " +
2933             (out.length - 1));
2934     }
2935     if (len > (out.length - offset)) {
2936         throw new IllegalArgumentException("input len is out of bound: " + len + " > " + (out.
2937             length - offset));
2938     }
2939 }
2940
2941 /**
2942  * Java Runtime may use intrinsic for this method.

```

```

2940     */
2941     private static int implMulAdd(int[] out, int[] in, int offset, int len, int k) {
2942         long kLong = k & LONG_MASK;
2943         long carry = 0;
2944
2945         offset = out.length - offset - 1;
2946         for (int j = len - 1; j >= 0; j--) {
2947             long product = (in[j] & LONG_MASK) * kLong +
2948                 (out[offset] & LONG_MASK) + carry;
2949             out[offset--] = (int)product;
2950             carry = product >>> 32;
2951         }
2952         return (int)carry;
2953     }
2954
2955     /**
2956     * Add one word to the number a mlen words into a. Return the resulting
2957     * carry.
2958     */
2959     static int addOne(int[] a, int offset, int mlen, int carry) {
2960         offset = a.length - 1 - mlen - offset;
2961         long t = (a[offset] & LONG_MASK) + (carry & LONG_MASK);
2962
2963         a[offset] = (int)t;
2964         if ((t >>> 32) == 0)
2965             return 0;
2966         while (--mlen >= 0) {
2967             if (--offset < 0) { // Carry out of number
2968                 return 1;
2969             } else {
2970                 a[offset]++;
2971                 if (a[offset] != 0)
2972                     return 0;
2973             }
2974         }
2975         return 1;
2976     }
2977
2978     /**
2979     * Returns a BigInteger whose value is (this ** exponent) mod (2**p)
2980     */
2981     private BigInteger modPow2(BigInteger exponent, int p) {
2982         /*
2983         * Perform exponentiation using repeated squaring trick, chopping off
2984         * high order bits as indicated by modulus.
2985         */
2986         BigInteger result = ONE;
2987         BigInteger baseToPow2 = this.mod2(p);
2988         int expOffset = 0;
2989
2990         int limit = exponent.bitLength();
2991
2992         if (this.testBit(0))

```

```

2993         limit = (p-1) < limit ? (p-1) : limit;
2994
2995         while (expOffset < limit) {
2996             if (exponent.testBit(expOffset))
2997                 result = result.multiply(baseToPow2).mod2(p);
2998             expOffset++;
2999             if (expOffset < limit)
3000                 baseToPow2 = baseToPow2.square().mod2(p);
3001         }
3002
3003         return result;
3004     }
3005
3006     /**
3007      * Returns a BigInteger whose value is this mod(2**p).
3008      * Assumes that this {@code BigInteger} >= 0 and {@code p} > 0.
3009      */
3010     private BigInteger mod2(int p) {
3011         if (bitLength() <= p)
3012             return this;
3013
3014         // Copy remaining ints of mag
3015         int numInts = (p + 31) >>> 5;
3016         int[] mag = new int[numInts];
3017         System.arraycopy(this.mag, (this.mag.length - numInts), mag, 0, numInts);
3018
3019         // Mask out any excess bits
3020         int excessBits = (numInts << 5) - p;
3021         mag[0] &= (1L << (32-excessBits)) - 1;
3022
3023         return (mag[0] == 0 ? new BigInteger(1, mag) : new BigInteger(mag, 1));
3024     }
3025
3026     /**
3027      * Returns a BigInteger whose value is  $(this)^{-1} \bmod m$ .
3028      *
3029      * @param m the modulus.
3030      * @return  $(this)^{-1} \bmod m$ .
3031      * @throws ArithmeticException  $m \leq 0$ , or this BigInteger
3032      *         has no multiplicative inverse mod m (that is, this BigInteger
3033      *         is not relatively prime to m).
3034      */
3035     public BigInteger modInverse(BigInteger m) {
3036         if (m.signum != 1)
3037             throw new ArithmeticException("BigInteger: modulus not positive");
3038
3039         if (m.equals(ONE))
3040             return ZERO;
3041
3042         // Calculate (this mod m)
3043         BigInteger modVal = this;
3044         if (signum < 0 || (this.compareMagnitude(m) >= 0))
3045             modVal = this.mod(m);

```

```

3046
3047     if (modVal.equals(ONE))
3048         return ONE;
3049
3050     MutableBigInteger a = new MutableBigInteger(modVal);
3051     MutableBigInteger b = new MutableBigInteger(m);
3052
3053     MutableBigInteger result = a.mutableModInverse(b);
3054     return result.toBigInteger(1);
3055 }
3056
3057 // Shift Operations
3058
3059 /**
3060  * Returns a BigInteger whose value is {@code (this << n)}.
3061  * The shift distance, {@code n}, may be negative, in which case
3062  * this method performs a right shift.
3063  * (Computes  $\text{floor}(\text{this} * 2^{\text{n}}$ )
3064  *
3065  * @param n shift distance, in bits.
3066  * @return {@code this << n}
3067  * @see #shiftRight
3068  */
3069 public BigInteger shiftLeft(int n) {
3070     if (signum == 0)
3071         return ZERO;
3072     if (n > 0) {
3073         return new BigInteger(shiftLeft(mag, n), signum);
3074     } else if (n == 0) {
3075         return this;
3076     } else {
3077         // Possible int overflow in (-n) is not a trouble,
3078         // because shiftRightImpl considers its argument unsigned
3079         return shiftRightImpl(-n);
3080     }
3081 }
3082
3083 /**
3084  * Returns a magnitude array whose value is {@code (mag << n)}.
3085  * The shift distance, {@code n}, is considered unsigned.
3086  * (Computes  $\text{this} * 2^{\text{n}}$ )
3087  *
3088  * @param mag magnitude, the most-significant int ({@code mag[0]}) must be non-zero.
3089  * @param n unsigned shift distance, in bits.
3090  * @return {@code mag << n}
3091  */
3092 private static int[] shiftLeft(int[] mag, int n) {
3093     int nInts = n >>> 5;
3094     int nBits = n & 0x1f;
3095     int magLen = mag.length;
3096     int newMag[] = null;
3097
3098     if (nBits == 0) {

```

```

3099         newMag = new int[magLen + nInts];
3100         System.arraycopy(mag, 0, newMag, 0, magLen);
3101     } else {
3102         int i = 0;
3103         int nBits2 = 32 - nBits;
3104         int highBits = mag[0] >>> nBits2;
3105         if (highBits != 0) {
3106             newMag = new int[magLen + nInts + 1];
3107             newMag[i++] = highBits;
3108         } else {
3109             newMag = new int[magLen + nInts];
3110         }
3111         int j=0;
3112         while (j < magLen-1)
3113             newMag[i++] = mag[j++] << nBits | mag[j] >>> nBits2;
3114         newMag[i] = mag[j] << nBits;
3115     }
3116     return newMag;
3117 }
3118
3119 /**
3120  * Returns a BigInteger whose value is {@code (this >> n)}. Sign
3121  * extension is performed. The shift distance, {@code n}, may be
3122  * negative, in which case this method performs a left shift.
3123  * (Computes  $\text{floor}(\text{this} / 2^{\text{n}})$ .)
3124  *
3125  * @param n shift distance, in bits.
3126  * @return {@code this >> n}
3127  * @see #shiftLeft
3128  */
3129 public BigInteger shiftRight(int n) {
3130     if (signum == 0)
3131         return ZERO;
3132     if (n > 0) {
3133         return shiftRightImpl(n);
3134     } else if (n == 0) {
3135         return this;
3136     } else {
3137         // Possible int overflow in {@code -n} is not a trouble,
3138         // because shiftLeft considers its argument unsigned
3139         return new BigInteger(shiftLeft(mag, -n), signum);
3140     }
3141 }
3142
3143 /**
3144  * Returns a BigInteger whose value is {@code (this >> n)}. The shift
3145  * distance, {@code n}, is considered unsigned.
3146  * (Computes  $\text{floor}(\text{this} * 2^{-\text{n}})$ .)
3147  *
3148  * @param n unsigned shift distance, in bits.
3149  * @return {@code this >> n}
3150  */
3151 private BigInteger shiftRightImpl(int n) {

```



```

3152     int nInts = n >>> 5;
3153     int nBits = n & 0x1f;
3154     int magLen = mag.length;
3155     int newMag[] = null;
3156
3157     // Special case: entire contents shifted off the end
3158     if (nInts >= magLen)
3159         return (signum >= 0 ? ZERO : negConst[1]);
3160
3161     if (nBits == 0) {
3162         int newMagLen = magLen - nInts;
3163         newMag = Arrays.copyOf(mag, newMagLen);
3164     } else {
3165         int i = 0;
3166         int highBits = mag[0] >>> nBits;
3167         if (highBits != 0) {
3168             newMag = new int[magLen - nInts];
3169             newMag[i++] = highBits;
3170         } else {
3171             newMag = new int[magLen - nInts - 1];
3172         }
3173
3174         int nBits2 = 32 - nBits;
3175         int j=0;
3176         while (j < magLen - nInts - 1)
3177             newMag[i++] = (mag[j++] << nBits2) | (mag[j] >>> nBits);
3178     }
3179
3180     if (signum < 0) {
3181         // Find out whether any one-bits were shifted off the end.
3182         boolean onesLost = false;
3183         for (int i=magLen-1, j=magLen-nInts; i >= j && !onesLost; i--)
3184             onesLost = (mag[i] != 0);
3185         if (!onesLost && nBits != 0)
3186             onesLost = (mag[magLen - nInts - 1] << (32 - nBits) != 0);
3187
3188         if (onesLost)
3189             newMag = javaIncrement(newMag);
3190     }
3191
3192     return new BigInteger(newMag, signum);
3193 }
3194
3195 int[] javaIncrement(int[] val) {
3196     int lastSum = 0;
3197     for (int i=val.length-1; i >= 0 && lastSum == 0; i--)
3198         lastSum = (val[i] += 1);
3199     if (lastSum == 0) {
3200         val = new int[val.length+1];
3201         val[0] = 1;
3202     }
3203     return val;
3204 }

```

```

3205
3206 // Bitwise Operations
3207
3208 /**
3209  * Returns a BigInteger whose value is {@code (this & val)}. (This
3210  * method returns a negative BigInteger if and only if this and val are
3211  * both negative.)
3212  *
3213  * @param val value to be AND'ed with this BigInteger.
3214  * @return {@code this & val}
3215  */
3216 public BigInteger and(BigInteger val) {
3217     int[] result = new int[Math.max(intLength(), val.intLength())];
3218     for (int i=0; i < result.length; i++)
3219         result[i] = (getInt(result.length-i-1)
3220             & val.getInt(result.length-i-1));
3221
3222     return valueOf(result);
3223 }
3224
3225 /**
3226  * Returns a BigInteger whose value is {@code (this | val)}. (This method
3227  * returns a negative BigInteger if and only if either this or val is
3228  * negative.)
3229  *
3230  * @param val value to be OR'ed with this BigInteger.
3231  * @return {@code this | val}
3232  */
3233 public BigInteger or(BigInteger val) {
3234     int[] result = new int[Math.max(intLength(), val.intLength())];
3235     for (int i=0; i < result.length; i++)
3236         result[i] = (getInt(result.length-i-1)
3237             | val.getInt(result.length-i-1));
3238
3239     return valueOf(result);
3240 }
3241
3242 /**
3243  * Returns a BigInteger whose value is {@code (this ^ val)}. (This method
3244  * returns a negative BigInteger if and only if exactly one of this and
3245  * val are negative.)
3246  *
3247  * @param val value to be XOR'ed with this BigInteger.
3248  * @return {@code this ^ val}
3249  */
3250 public BigInteger xor(BigInteger val) {
3251     int[] result = new int[Math.max(intLength(), val.intLength())];
3252     for (int i=0; i < result.length; i++)
3253         result[i] = (getInt(result.length-i-1)
3254             ^ val.getInt(result.length-i-1));
3255
3256     return valueOf(result);
3257 }

```

```

3258
3259 /**
3260  * Returns a BigInteger whose value is {@code (~this)}. (This method
3261  * returns a negative value if and only if this BigInteger is
3262  * non-negative.)
3263  *
3264  * @return {@code ~this}
3265  */
3266 public BigInteger not() {
3267     int[] result = new int[intLength()];
3268     for (int i=0; i < result.length; i++)
3269         result[i] = ~getInt(result.length-i-1);
3270
3271     return valueOf(result);
3272 }
3273
3274 /**
3275  * Returns a BigInteger whose value is {@code (this & ~val)}. This
3276  * method, which is equivalent to {@code and(val.not())}, is provided as
3277  * a convenience for masking operations. (This method returns a negative
3278  * BigInteger if and only if {@code this} is negative and {@code val} is
3279  * positive.)
3280  *
3281  * @param val value to be complemented and AND'ed with this BigInteger.
3282  * @return {@code this & ~val}
3283  */
3284 public BigInteger andNot(BigInteger val) {
3285     int[] result = new int[Math.max(intLength(), val.intLength())];
3286     for (int i=0; i < result.length; i++)
3287         result[i] = (getInt(result.length-i-1)
3288                     & ~val.getInt(result.length-i-1));
3289
3290     return valueOf(result);
3291 }
3292
3293
3294 // Single Bit Operations
3295
3296 /**
3297  * Returns {@code true} if and only if the designated bit is set.
3298  * (Computes {@code ((this & (1<<n)) != 0)}.)
3299  *
3300  * @param n index of bit to test.
3301  * @return {@code true} if and only if the designated bit is set.
3302  * @throws ArithmeticException {@code n} is negative.
3303  */
3304 public boolean testBit(int n) {
3305     if (n < 0)
3306         throw new ArithmeticException("Negative bit address");
3307
3308     return (getInt(n >>> 5) & (1 << (n & 31))) != 0;
3309 }
3310

```

```

3311 /**
3312  * Returns a BigInteger whose value is equivalent to this BigInteger
3313  * with the designated bit set. (Computes {@code (this | (1<<n))}.)
3314  *
3315  * @param n index of bit to set.
3316  * @return {@code this | (1<<n)}
3317  * @throws ArithmeticException {@code n} is negative.
3318  */
3319 public BigInteger setBit(int n) {
3320     if (n < 0)
3321         throw new ArithmeticException("Negative bit address");
3322
3323     int intNum = n >>> 5;
3324     int[] result = new int[Math.max(intLength(), intNum+2)];
3325
3326     for (int i=0; i < result.length; i++)
3327         result[result.length-i-1] = getInt(i);
3328
3329     result[result.length-intNum-1] |= (1 << (n & 31));
3330
3331     return valueOf(result);
3332 }
3333
3334 /**
3335  * Returns a BigInteger whose value is equivalent to this BigInteger
3336  * with the designated bit cleared.
3337  * (Computes {@code (this & ~(1<<n))}.)
3338  *
3339  * @param n index of bit to clear.
3340  * @return {@code this & ~(1<<n)}
3341  * @throws ArithmeticException {@code n} is negative.
3342  */
3343 public BigInteger clearBit(int n) {
3344     if (n < 0)
3345         throw new ArithmeticException("Negative bit address");
3346
3347     int intNum = n >>> 5;
3348     int[] result = new int[Math.max(intLength(), ((n + 1) >>> 5) + 1)];
3349
3350     for (int i=0; i < result.length; i++)
3351         result[result.length-i-1] = getInt(i);
3352
3353     result[result.length-intNum-1] &= ~(1 << (n & 31));
3354
3355     return valueOf(result);
3356 }
3357
3358 /**
3359  * Returns a BigInteger whose value is equivalent to this BigInteger
3360  * with the designated bit flipped.
3361  * (Computes {@code (this ^ (1<<n))}.)
3362  *
3363  * @param n index of bit to flip.

```

```

3364     * @return {@code this ^ (1<<n)}
3365     * @throws ArithmeticException {@code n} is negative.
3366     */
3367     public BigInteger flipBit(int n) {
3368         if (n < 0)
3369             throw new ArithmeticException("Negative bit address");
3370
3371         int intNum = n >>> 5;
3372         int[] result = new int[Math.max(intLength(), intNum+2)];
3373
3374         for (int i=0; i < result.length; i++)
3375             result[result.length-i-1] = getInt(i);
3376
3377         result[result.length-intNum-1] ^= (1 << (n & 31));
3378
3379         return valueOf(result);
3380     }
3381
3382     /**
3383     * Returns the index of the rightmost (lowest-order) one bit in this
3384     * BigInteger (the number of zero bits to the right of the rightmost
3385     * one bit). Returns -1 if this BigInteger contains no one bits.
3386     * (Computes {@code (this == 0 ? -1 : log2(this & -this))}.)
3387     *
3388     * @return index of the rightmost one bit in this BigInteger.
3389     */
3390     public int getLowestSetBit() {
3391         @SuppressWarnings("deprecation") int lsb = lowestSetBit - 2;
3392         if (lsb == -2) { // lowestSetBit not initialized yet
3393             lsb = 0;
3394             if (signum == 0) {
3395                 lsb -= 1;
3396             } else {
3397                 // Search for lowest order nonzero int
3398                 int i,b;
3399                 for (i=0; (b = getInt(i)) == 0; i++)
3400                     ;
3401                 lsb += (i << 5) + Integer.numberOfTrailingZeros(b);
3402             }
3403             lowestSetBit = lsb + 2;
3404         }
3405         return lsb;
3406     }
3407
3408
3409     // Miscellaneous Bit Operations
3410
3411     /**
3412     * Returns the number of bits in the minimal two's-complement
3413     * representation of this BigInteger, <i>excluding</i> a sign bit.
3414     * For positive BigIntegers, this is equivalent to the number of bits in
3415     * the ordinary binary representation. (Computes
3416     * {@code (ceil(log2(this < 0 ? -this : this+1)))}.)

```

```

3417 *
3418 * @return number of bits in the minimal two's-complement
3419 * representation of this BigInteger, <i>excluding</i> a sign bit.
3420 */
3421 public int bitLength() {
3422     @SuppressWarnings("deprecation") int n = bitLength - 1;
3423     if (n == -1) { // bitLength not initialized yet
3424         int[] m = mag;
3425         int len = m.length;
3426         if (len == 0) {
3427             n = 0; // offset by one to initialize
3428         } else {
3429             // Calculate the bit length of the magnitude
3430             int magBitLength = ((len - 1) << 5) + bitLengthForInt(mag[0]);
3431             if (signum < 0) {
3432                 // Check if magnitude is a power of two
3433                 boolean pow2 = (Integer.bitCount(mag[0]) == 1);
3434                 for (int i=1; i< len && pow2; i++)
3435                     pow2 = (mag[i] == 0);
3436
3437                 n = (pow2 ? magBitLength - 1 : magBitLength);
3438             } else {
3439                 n = magBitLength;
3440             }
3441         }
3442         bitLength = n + 1;
3443     }
3444     return n;
3445 }
3446
3447 /**
3448 * Returns the number of bits in the two's complement representation
3449 * of this BigInteger that differ from its sign bit. This method is
3450 * useful when implementing bit-vector style sets atop BigIntegers.
3451 *
3452 * @return number of bits in the two's complement representation
3453 * of this BigInteger that differ from its sign bit.
3454 */
3455 public int bitCount() {
3456     @SuppressWarnings("deprecation") int bc = bitCount - 1;
3457     if (bc == -1) { // bitCount not initialized yet
3458         bc = 0; // offset by one to initialize
3459         // Count the bits in the magnitude
3460         for (int i=0; i < mag.length; i++)
3461             bc += Integer.bitCount(mag[i]);
3462         if (signum < 0) {
3463             // Count the trailing zeros in the magnitude
3464             int magTrailingZeroCount = 0, j;
3465             for (j=mag.length-1; mag[j] == 0; j--)
3466                 magTrailingZeroCount += 32;
3467             magTrailingZeroCount += Integer.numberOfTrailingZeros(mag[j]);
3468             bc += magTrailingZeroCount - 1;
3469         }

```

```

3470         bitCount = bc + 1;
3471     }
3472     return bc;
3473 }
3474
3475 // Primality Testing
3476
3477 /**
3478  * Returns {@code true} if this BigInteger is probably prime,
3479  * {@code false} if it's definitely composite. If
3480  * {@code certainty} is  $\leq 0$ , {@code true} is
3481  * returned.
3482  *
3483  * @param certainty a measure of the uncertainty that the caller is
3484  *        willing to tolerate: if the call returns {@code true}
3485  *        the probability that this BigInteger is prime exceeds
3486  *         $(1 - 1/2^{\text{certainty}})$ . The execution time of
3487  *        this method is proportional to the value of this parameter.
3488  * @return {@code true} if this BigInteger is probably prime,
3489  *        {@code false} if it's definitely composite.
3490  */
3491 public boolean isProbablePrime(int certainty) {
3492     if (certainty <= 0)
3493         return true;
3494     BigInteger w = this.abs();
3495     if (w.equals(TWO))
3496         return true;
3497     if (!w.testBit(0) || w.equals(ONE))
3498         return false;
3499
3500     return w.primeToCertainty(certainty, null);
3501 }
3502
3503 // Comparison Operations
3504
3505 /**
3506  * Compares this BigInteger with the specified BigInteger. This
3507  * method is provided in preference to individual methods for each
3508  * of the six boolean comparison operators ({@literal <}, ==,
3509  * {@literal >}, {@literal >=}, !=, {@literal <=}). The suggested
3510  * idiom for performing these comparisons is: {@code
3511  * (x.compareTo(y)) <i>op</i> {code 0}}, where
3512  * <i>op</i> is one of the six comparison operators.
3513  *
3514  * @param val BigInteger to which this BigInteger is to be compared.
3515  * @return -1, 0 or 1 as this BigInteger is numerically less than, equal
3516  *         to, or greater than {@code val}.
3517  */
3518 public int compareTo(BigInteger val) {
3519     if (signum == val.signum) {
3520         switch (signum) {
3521             case 1:
3522                 return compareMagnitude(val);

```

```

3523         case -1:
3524             return val.compareMagnitude(this);
3525         default:
3526             return 0;
3527     }
3528 }
3529 return signum > val.signum ? 1 : -1;
3530 }
3531
3532 /**
3533  * Compares the magnitude array of this BigInteger with the specified
3534  * BigInteger's. This is the version of compareTo ignoring sign.
3535  *
3536  * @param val BigInteger whose magnitude array to be compared.
3537  * @return -1, 0 or 1 as this magnitude array is less than, equal to or
3538  *         greater than the magnitude array for the specified BigInteger's.
3539  */
3540 final int compareMagnitude(BigInteger val) {
3541     int[] m1 = mag;
3542     int len1 = m1.length;
3543     int[] m2 = val.mag;
3544     int len2 = m2.length;
3545     if (len1 < len2)
3546         return -1;
3547     if (len1 > len2)
3548         return 1;
3549     for (int i = 0; i < len1; i++) {
3550         int a = m1[i];
3551         int b = m2[i];
3552         if (a != b)
3553             return ((a & LONG_MASK) < (b & LONG_MASK)) ? -1 : 1;
3554     }
3555     return 0;
3556 }
3557
3558 /**
3559  * Version of compareMagnitude that compares magnitude with long value.
3560  * val can't be Long.MIN_VALUE.
3561  */
3562 final int compareMagnitude(long val) {
3563     assert val != Long.MIN_VALUE;
3564     int[] m1 = mag;
3565     int len = m1.length;
3566     if (len > 2) {
3567         return 1;
3568     }
3569     if (val < 0) {
3570         val = -val;
3571     }
3572     int highWord = (int)(val >>> 32);
3573     if (highWord == 0) {
3574         if (len < 1)
3575             return -1;

```



```

3576         if (len > 1)
3577             return 1;
3578         int a = m1[0];
3579         int b = (int)val;
3580         if (a != b) {
3581             return ((a & LONG_MASK) < (b & LONG_MASK)) ? -1 : 1;
3582         }
3583         return 0;
3584     } else {
3585         if (len < 2)
3586             return -1;
3587         int a = m1[0];
3588         int b = highWord;
3589         if (a != b) {
3590             return ((a & LONG_MASK) < (b & LONG_MASK)) ? -1 : 1;
3591         }
3592         a = m1[1];
3593         b = (int)val;
3594         if (a != b) {
3595             return ((a & LONG_MASK) < (b & LONG_MASK)) ? -1 : 1;
3596         }
3597         return 0;
3598     }
3599 }
3600
3601 /**
3602  * Compares this BigInteger with the specified Object for equality.
3603  *
3604  * @param x Object to which this BigInteger is to be compared.
3605  * @return {@code true} if and only if the specified Object is a
3606  *         BigInteger whose value is numerically equal to this BigInteger.
3607  */
3608 public boolean equals(Object x) {
3609     // This test is just an optimization, which may or may not help
3610     if (x == this)
3611         return true;
3612
3613     if (!(x instanceof BigInteger))
3614         return false;
3615
3616     BigInteger xInt = (BigInteger) x;
3617     if (xInt.signum != signum)
3618         return false;
3619
3620     int[] m = mag;
3621     int len = m.length;
3622     int[] xm = xInt.mag;
3623     if (len != xm.length)
3624         return false;
3625
3626     for (int i = 0; i < len; i++)
3627         if (xm[i] != m[i])
3628             return false;

```

```

3629
3630     return true;
3631 }
3632
3633 /**
3634  * Returns the minimum of this BigInteger and {@code val}.
3635  *
3636  * @param val value with which the minimum is to be computed.
3637  * @return the BigInteger whose value is the lesser of this BigInteger and
3638  *         {@code val}. If they are equal, either may be returned.
3639  */
3640 public BigInteger min(BigInteger val) {
3641     return (compareTo(val) < 0 ? this : val);
3642 }
3643
3644 /**
3645  * Returns the maximum of this BigInteger and {@code val}.
3646  *
3647  * @param val value with which the maximum is to be computed.
3648  * @return the BigInteger whose value is the greater of this and
3649  *         {@code val}. If they are equal, either may be returned.
3650  */
3651 public BigInteger max(BigInteger val) {
3652     return (compareTo(val) > 0 ? this : val);
3653 }
3654
3655
3656 // Hash Function
3657
3658 /**
3659  * Returns the hash code for this BigInteger.
3660  *
3661  * @return hash code for this BigInteger.
3662  */
3663 public int hashCode() {
3664     int hashCode = 0;
3665
3666     for (int i=0; i < mag.length; i++)
3667         hashCode = (int)(31*hashCode + (mag[i] & LONG_MASK));
3668
3669     return hashCode * signum;
3670 }
3671
3672 /**
3673  * Returns the String representation of this BigInteger in the
3674  * given radix. If the radix is outside the range from {@link
3675  * Character#MIN_RADIX} to {@link Character#MAX_RADIX} inclusive,
3676  * it will default to 10 (as is the case for
3677  * {@code Integer.toString}). The digit-to-character mapping
3678  * provided by {@code Character.forDigit} is used, and a minus
3679  * sign is prepended if appropriate. (This representation is
3680  * compatible with the {@link #BigInteger(String, int) (String,
3681  * int)} constructor.)

```

```

3682     *
3683     * @param radix radix of the String representation.
3684     * @return String representation of this BigInteger in the given radix.
3685     * @see Integer#toString
3686     * @see Character#forDigit
3687     * @see #BigInteger(java.lang.String, int)
3688     */
3689     public String toString(int radix) {
3690         if (signum == 0)
3691             return "0";
3692         if (radix < Character.MIN_RADIX || radix > Character.MAX_RADIX)
3693             radix = 10;
3694
3695         // If it's small enough, use smallToString.
3696         if (mag.length <= SCHOENHAGE_BASE_CONVERSION_THRESHOLD)
3697             return smallToString(radix);
3698
3699         // Otherwise use recursive toString, which requires positive arguments.
3700         // The results will be concatenated into this StringBuilder
3701         StringBuilder sb = new StringBuilder();
3702         if (signum < 0) {
3703             toString(this.negate(), sb, radix, 0);
3704             sb.insert(0, '-');
3705         }
3706         else
3707             toString(this, sb, radix, 0);
3708
3709         return sb.toString();
3710     }
3711
3712     /** This method is used to perform toString when arguments are small. */
3713     private String smallToString(int radix) {
3714         if (signum == 0) {
3715             return "0";
3716         }
3717
3718         // Compute upper bound on number of digit groups and allocate space
3719         int maxNumDigitGroups = (4*mag.length + 6)/7;
3720         String digitGroup[] = new String[maxNumDigitGroups];
3721
3722         // Translate number to string, a digit group at a time
3723         BigInteger tmp = this.abs();
3724         int numGroups = 0;
3725         while (tmp.signum != 0) {
3726             BigInteger d = longRadix[radix];
3727
3728             MutableBigInteger q = new MutableBigInteger(),
3729                 a = new MutableBigInteger(tmp.mag),
3730                 b = new MutableBigInteger(d.mag);
3731             MutableBigInteger r = a.divide(b, q);
3732             BigInteger q2 = q.toBigInteger(tmp.signum * d.signum);
3733             BigInteger r2 = r.toBigInteger(tmp.signum * d.signum);
3734

```

```

3735         digitGroup[numGroups++] = Long.toString(r2.longValue(), radix);
3736         tmp = q2;
3737     }
3738
3739     // Put sign (if any) and first digit group into result buffer
3740     StringBuilder buf = new StringBuilder(numGroups*digitsPerLong[radix]+1);
3741     if (signum < 0) {
3742         buf.append('-');
3743     }
3744     buf.append(digitGroup[numGroups-1]);
3745
3746     // Append remaining digit groups padded with leading zeros
3747     for (int i=numGroups-2; i >= 0; i--) {
3748         // Prepend (any) leading zeros for this digit group
3749         int numLeadingZeros = digitsPerLong[radix]-digitGroup[i].length();
3750         if (numLeadingZeros != 0) {
3751             buf.append(zeros[numLeadingZeros]);
3752         }
3753         buf.append(digitGroup[i]);
3754     }
3755     return buf.toString();
3756 }
3757
3758 /**
3759  * Converts the specified BigInteger to a string and appends to
3760  * {@code sb}. This implements the recursive Schoenhage algorithm
3761  * for base conversions.
3762  * <p>
3763  * See Knuth, Donald, The Art of Computer Programming, Vol. 2,
3764  * Answers to Exercises (4.4) Question 14.
3765  *
3766  * @param u      The number to convert to a string.
3767  * @param sb      The StringBuilder that will be appended to in place.
3768  * @param radix   The base to convert to.
3769  * @param digits  The minimum number of digits to pad to.
3770  */
3771 private static void toString(BigInteger u, StringBuilder sb, int radix,
3772                             int digits) {
3773     /* If we're smaller than a certain threshold, use the smallToString
3774      method, padding with leading zeroes when necessary. */
3775     if (u.mag.length <= SCHOENHAGE_BASE_CONVERSION_THRESHOLD) {
3776         String s = u.smallToString(radix);
3777
3778         // Pad with internal zeros if necessary.
3779         // Don't pad if we're at the beginning of the string.
3780         if ((s.length() < digits) && (sb.length() > 0)) {
3781             for (int i=s.length(); i < digits; i++) { // May be a faster way to
3782                 sb.append('0');                      // do this?
3783             }
3784         }
3785
3786         sb.append(s);
3787         return;

```

[illegible]

```

3841     for (int i=0; i < 63; i++)
3842         zeros[i] = zeros[63].substring(0, i);
3843     }
3844
3845     /**
3846     * Returns the decimal String representation of this BigInteger.
3847     * The digit-to-character mapping provided by
3848     * {@code Character.forDigit} is used, and a minus sign is
3849     * prepended if appropriate. (This representation is compatible
3850     * with the {@link #BigInteger(String) (String)} constructor, and
3851     * allows for String concatenation with Java's + operator.)
3852     *
3853     * @return decimal String representation of this BigInteger.
3854     * @see    Character#forDigit
3855     * @see    #BigInteger(java.lang.String)
3856     */
3857     public String toString() {
3858         return toString(10);
3859     }
3860
3861     /**
3862     * Returns a byte array containing the two's-complement
3863     * representation of this BigInteger. The byte array will be in
3864     * <i>big-endian</i> byte-order: the most significant byte is in
3865     * the zeroth element. The array will contain the minimum number
3866     * of bytes required to represent this BigInteger, including at
3867     * least one sign bit, which is {@code (ceil((this.bitLength() +
3868     * 1)/8))}. (This representation is compatible with the
3869     * {@link #BigInteger(byte[]) (byte[])} constructor.)
3870     *
3871     * @return a byte array containing the two's-complement representation of
3872     *         this BigInteger.
3873     * @see    #BigInteger(byte[])
3874     */
3875     public byte[] toByteArray() {
3876         int byteLen = bitLength()/8 + 1;
3877         byte[] byteArray = new byte[byteLen];
3878
3879         for (int i=byteLen-1, bytesCopied=4, nextInt=0, intIndex=0; i >= 0; i--) {
3880             if (bytesCopied == 4) {
3881                 nextInt = getInt(intIndex++);
3882                 bytesCopied = 1;
3883             } else {
3884                 nextInt >>= 8;
3885                 bytesCopied++;
3886             }
3887             byteArray[i] = (byte)nextInt;
3888         }
3889         return byteArray;
3890     }
3891
3892     /**
3893     * Converts this BigInteger to an {@code int}. This

```

```

3894     * conversion is analogous to a
3895     * <i>narrowing primitive conversion</i> from {@code long} to
3896     * {@code int} as defined in section 5.1.3 of
3897     * <cite>The Java&trade; Language Specification</cite>:
3898     * if this BigInteger is too big to fit in an
3899     * {@code int}, only the low-order 32 bits are returned.
3900     * Note that this conversion can lose information about the
3901     * overall magnitude of the BigInteger value as well as return a
3902     * result with the opposite sign.
3903     *
3904     * @return this BigInteger converted to an {@code int}.
3905     * @see #intValueExact()
3906     */
3907     public int intValue() {
3908         int result = 0;
3909         result = getInt(0);
3910         return result;
3911     }
3912
3913     /**
3914     * Converts this BigInteger to a {@code long}. This
3915     * conversion is analogous to a
3916     * <i>narrowing primitive conversion</i> from {@code long} to
3917     * {@code int} as defined in section 5.1.3 of
3918     * <cite>The Java&trade; Language Specification</cite>:
3919     * if this BigInteger is too big to fit in a
3920     * {@code long}, only the low-order 64 bits are returned.
3921     * Note that this conversion can lose information about the
3922     * overall magnitude of the BigInteger value as well as return a
3923     * result with the opposite sign.
3924     *
3925     * @return this BigInteger converted to a {@code long}.
3926     * @see #longValueExact()
3927     */
3928     public long longValue() {
3929         long result = 0;
3930
3931         for (int i=1; i >= 0; i--)
3932             result = (result << 32) + (getInt(i) & LONG_MASK);
3933         return result;
3934     }
3935
3936     /**
3937     * Converts this BigInteger to a {@code float}. This
3938     * conversion is similar to the
3939     * <i>narrowing primitive conversion</i> from {@code double} to
3940     * {@code float} as defined in section 5.1.3 of
3941     * <cite>The Java&trade; Language Specification</cite>:
3942     * if this BigInteger has too great a magnitude
3943     * to represent as a {@code float}, it will be converted to
3944     * {@link Float#NEGATIVE_INFINITY} or {@link
3945     * Float#POSITIVE_INFINITY} as appropriate. Note that even when
3946     * the return value is finite, this conversion can lose

```

```

3947     * information about the precision of the BigInteger value.
3948     *
3949     * @return this BigInteger converted to a {@code float}.
3950     */
3951     public float floatValue() {
3952         if (signum == 0) {
3953             return 0.0f;
3954         }
3955
3956         int exponent = ((mag.length - 1) << 5) + bitLengthForInt(mag[0]) - 1;
3957
3958         // exponent == floor(log2(abs(this)))
3959         if (exponent < Long.SIZE - 1) {
3960             return longValue();
3961         } else if (exponent > Float.MAX_EXPONENT) {
3962             return signum > 0 ? Float.POSITIVE_INFINITY : Float.NEGATIVE_INFINITY;
3963         }
3964
3965         /*
3966          * We need the top SIGNIFICAND_WIDTH bits, including the "implicit"
3967          * one bit. To make rounding easier, we pick out the top
3968          * SIGNIFICAND_WIDTH + 1 bits, so we have one to help us round up or
3969          * down. twiceSignifFloor will contain the top SIGNIFICAND_WIDTH + 1
3970          * bits, and signifFloor the top SIGNIFICAND_WIDTH.
3971          *
3972          * It helps to consider the real number signif = abs(this) *
3973          * 2^(SIGNIFICAND_WIDTH - 1 - exponent).
3974          */
3975         int shift = exponent - FloatConsts.SIGNIFICAND_WIDTH;
3976
3977         int twiceSignifFloor;
3978         // twiceSignifFloor will be == abs().shiftRight(shift).intValue()
3979         // We do the shift into an int directly to improve performance.
3980
3981         int nBits = shift & 0x1f;
3982         int nBits2 = 32 - nBits;
3983
3984         if (nBits == 0) {
3985             twiceSignifFloor = mag[0];
3986         } else {
3987             twiceSignifFloor = mag[0] >>> nBits;
3988             if (twiceSignifFloor == 0) {
3989                 twiceSignifFloor = (mag[0] << nBits2) | (mag[1] >>> nBits);
3990             }
3991         }
3992
3993         int signifFloor = twiceSignifFloor >> 1;
3994         signifFloor &= FloatConsts.SIGNIF_BIT_MASK; // remove the implied bit
3995
3996         /*
3997          * We round up if either the fractional part of signif is strictly
3998          * greater than 0.5 (which is true if the 0.5 bit is set and any lower
3999          * bit is set), or if the fractional part of signif is >= 0.5 and

```



```

4000     * signifFloor is odd (which is true if both the 0.5 bit and the 1 bit
4001     * are set). This is equivalent to the desired HALF_EVEN rounding.
4002     */
4003     boolean increment = (twiceSignifFloor & 1) != 0
4004         && ((signifFloor & 1) != 0 || abs().getLowestSetBit() < shift);
4005     int signifRounded = increment ? signifFloor + 1 : signifFloor;
4006     int bits = ((exponent + FloatConsts.EXP_BIAS))
4007         << (FloatConsts.SIGNIFICAND_WIDTH - 1);
4008     bits += signifRounded;
4009     /*
4010     * If signifRounded == 2^24, we'd need to set all of the significand
4011     * bits to zero and add 1 to the exponent. This is exactly the behavior
4012     * we get from just adding signifRounded to bits directly. If the
4013     * exponent is Float.MAX_EXPONENT, we round up (correctly) to
4014     * Float.POSITIVE_INFINITY.
4015     */
4016     bits |= signum & FloatConsts.SIGN_BIT_MASK;
4017     return Float.intBitsToFloat(bits);
4018 }
4019
4020 /**
4021  * Converts this BigInteger to a {@code double}. This
4022  * conversion is similar to the
4023  * <i>narrowing primitive conversion</i> from {@code double} to
4024  * {@code float} as defined in section 5.1.3 of
4025  * <cite>The Java&trade; Language Specification</cite>:
4026  * if this BigInteger has too great a magnitude
4027  * to represent as a {@code double}, it will be converted to
4028  * {@link Double#NEGATIVE_INFINITY} or {@link
4029  * Double#POSITIVE_INFINITY} as appropriate. Note that even when
4030  * the return value is finite, this conversion can lose
4031  * information about the precision of the BigInteger value.
4032  *
4033  * @return this BigInteger converted to a {@code double}.
4034  */
4035 public double doubleValue() {
4036     if (signum == 0) {
4037         return 0.0;
4038     }
4039
4040     int exponent = ((mag.length - 1) << 5) + bitLengthForInt(mag[0]) - 1;
4041
4042     // exponent == floor(log2(abs(this))Double)
4043     if (exponent < Long.SIZE - 1) {
4044         return longValue();
4045     } else if (exponent > Double.MAX_EXPONENT) {
4046         return signum > 0 ? Double.POSITIVE_INFINITY : Double.NEGATIVE_INFINITY;
4047     }
4048
4049     /*
4050     * We need the top SIGNIFICAND_WIDTH bits, including the "implicit"
4051     * one bit. To make rounding easier, we pick out the top
4052     * SIGNIFICAND_WIDTH + 1 bits, so we have one to help us round up or

```

```

4053     * down. twiceSignifFloor will contain the top SIGNIFICAND_WIDTH + 1
4054     * bits, and signifFloor the top SIGNIFICAND_WIDTH.
4055     *
4056     * It helps to consider the real number signif = abs(this) *
4057     * 2^(SIGNIFICAND_WIDTH - 1 - exponent).
4058     */
4059     int shift = exponent - DoubleConsts.SIGNIFICAND_WIDTH;
4060
4061     long twiceSignifFloor;
4062     // twiceSignifFloor will be == abs().shiftRight(shift).longValue()
4063     // We do the shift into a long directly to improve performance.
4064
4065     int nBits = shift & 0x1f;
4066     int nBits2 = 32 - nBits;
4067
4068     int highBits;
4069     int lowBits;
4070     if (nBits == 0) {
4071         highBits = mag[0];
4072         lowBits = mag[1];
4073     } else {
4074         highBits = mag[0] >>> nBits;
4075         lowBits = (mag[0] << nBits2) | (mag[1] >>> nBits);
4076         if (highBits == 0) {
4077             highBits = lowBits;
4078             lowBits = (mag[1] << nBits2) | (mag[2] >>> nBits);
4079         }
4080     }
4081
4082     twiceSignifFloor = ((highBits & LONG_MASK) << 32)
4083         | (lowBits & LONG_MASK);
4084
4085     long signifFloor = twiceSignifFloor >> 1;
4086     signifFloor &= DoubleConsts.SIGNIF_BIT_MASK; // remove the implied bit
4087
4088     /*
4089     * We round up if either the fractional part of signif is strictly
4090     * greater than 0.5 (which is true if the 0.5 bit is set and any lower
4091     * bit is set), or if the fractional part of signif is >= 0.5 and
4092     * signifFloor is odd (which is true if both the 0.5 bit and the 1 bit
4093     * are set). This is equivalent to the desired HALF_EVEN rounding.
4094     */
4095     boolean increment = (twiceSignifFloor & 1) != 0
4096         && ((signifFloor & 1) != 0 || abs().getLowestSetBit() < shift);
4097     long signifRounded = increment ? signifFloor + 1 : signifFloor;
4098     long bits = (long) ((exponent + DoubleConsts.EXP_BIAS)
4099         << (DoubleConsts.SIGNIFICAND_WIDTH - 1));
4100     bits += signifRounded;
4101     /*
4102     * If signifRounded == 2^53, we'd need to set all of the significand
4103     * bits to zero and add 1 to the exponent. This is exactly the behavior
4104     * we get from just adding signifRounded to bits directly. If the
4105     * exponent is Double.MAX_EXPONENT, we round up (correctly) to

```

```

4106         * Double.POSITIVE_INFINITY.
4107         */
4108         bits |= signum & DoubleConsts.SIGN_BIT_MASK;
4109         return Double.longBitsToDouble(bits);
4110     }
4111
4112     /**
4113      * Returns a copy of the input array stripped of any leading zero bytes.
4114      */
4115     private static int[] stripLeadingZeroInts(int val[]) {
4116         int vlen = val.length;
4117         int keep;
4118
4119         // Find first nonzero byte
4120         for (keep = 0; keep < vlen && val[keep] == 0; keep++)
4121             ;
4122         return java.util.Arrays.copyOfRange(val, keep, vlen);
4123     }
4124
4125     /**
4126      * Returns the input array stripped of any leading zero bytes.
4127      * Since the source is trusted the copying may be skipped.
4128      */
4129     private static int[] trustedStripLeadingZeroInts(int val[]) {
4130         int vlen = val.length;
4131         int keep;
4132
4133         // Find first nonzero byte
4134         for (keep = 0; keep < vlen && val[keep] == 0; keep++)
4135             ;
4136         return keep == 0 ? val : java.util.Arrays.copyOfRange(val, keep, vlen);
4137     }
4138
4139     /**
4140      * Returns a copy of the input array stripped of any leading zero bytes.
4141      */
4142     private static int[] stripLeadingZeroBytes(byte a[]) {
4143         int byteLength = a.length;
4144         int keep;
4145
4146         // Find first nonzero byte
4147         for (keep = 0; keep < byteLength && a[keep] == 0; keep++)
4148             ;
4149
4150         // Allocate new array and copy relevant part of input array
4151         int intLength = ((byteLength - keep) + 3) >>> 2;
4152         int[] result = new int[intLength];
4153         int b = byteLength - 1;
4154         for (int i = intLength-1; i >= 0; i--) {
4155             result[i] = a[b--] & 0xff;
4156             int bytesRemaining = b - keep + 1;
4157             int bytesToTransfer = Math.min(3, bytesRemaining);
4158             for (int j=8; j <= (bytesToTransfer << 3); j += 8)

```

```

4159         result[i] |= ((a[b--] & 0xff) << j);
4160     }
4161     return result;
4162 }
4163
4164 /**
4165  * Takes an array a representing a negative 2's-complement number and
4166  * returns the minimal (no leading zero bytes) unsigned whose value is -a.
4167  */
4168 private static int[] makePositive(byte a[]) {
4169     int keep, k;
4170     int byteLength = a.length;
4171
4172     // Find first non-sign (0xff) byte of input
4173     for (keep=0; keep < byteLength && a[keep] == -1; keep++)
4174         ;
4175
4176     /* Allocate output array. If all non-sign bytes are 0x00, we must
4177      * allocate space for one extra output byte. */
4178     for (k=keep; k < byteLength && a[k] == 0; k++)
4179         ;
4180
4181     int extraByte = (k == byteLength) ? 1 : 0;
4182     int intLength = ((byteLength - keep + extraByte) + 3) >>> 2;
4183     int result[] = new int[intLength];
4184
4185     /* Copy one's complement of input into output, leaving extra
4186      * byte (if it exists) == 0x00 */
4187     int b = byteLength - 1;
4188     for (int i = intLength-1; i >= 0; i--) {
4189         result[i] = a[b--] & 0xff;
4190         int numBytesToTransfer = Math.min(3, b-keep+1);
4191         if (numBytesToTransfer < 0)
4192             numBytesToTransfer = 0;
4193         for (int j=8; j <= 8*numBytesToTransfer; j += 8)
4194             result[i] |= ((a[b--] & 0xff) << j);
4195
4196         // Mask indicates which bits must be complemented
4197         int mask = -1 >>> (8*(3-numBytesToTransfer));
4198         result[i] = ~result[i] & mask;
4199     }
4200
4201     // Add one to one's complement to generate two's complement
4202     for (int i=result.length-1; i >= 0; i--) {
4203         result[i] = (int)((result[i] & LONG_MASK) + 1);
4204         if (result[i] != 0)
4205             break;
4206     }
4207
4208     return result;
4209 }
4210
4211

```

```

4212  /**
4213   * Takes an array a representing a negative 2's-complement number and
4214   * returns the minimal (no leading zero ints) unsigned whose value is -a.
4215   */
4216  private static int[] makePositive(int a[]) {
4217      int keep, j;
4218
4219      // Find first non-sign (0xffffffff) int of input
4220      for (keep=0; keep < a.length && a[keep] == -1; keep++)
4221          ;
4222
4223      /* Allocate output array. If all non-sign ints are 0x00, we must
4224       * allocate space for one extra output int. */
4225      for (j=keep; j < a.length && a[j] == 0; j++)
4226          ;
4227      int extraInt = (j == a.length ? 1 : 0);
4228      int result[] = new int[a.length - keep + extraInt];
4229
4230      /* Copy one's complement of input into output, leaving extra
4231       * int (if it exists) == 0x00 */
4232      for (int i = keep; i < a.length; i++)
4233          result[i - keep + extraInt] = ~a[i];
4234
4235      // Add one to one's complement to generate two's complement
4236      for (int i=result.length-1; ++result[i] == 0; i--)
4237          ;
4238
4239      return result;
4240  }
4241
4242  /**
4243   * The following two arrays are used for fast String conversions. Both
4244   * are indexed by radix. The first is the number of digits of the given
4245   * radix that can fit in a Java long without "going negative", i.e., the
4246   * highest integer n such that radix**n < 2**63. The second is the
4247   * "long radix" that tears each number into "long digits", each of which
4248   * consists of the number of digits in the corresponding element in
4249   * digitsPerLong (longRadix[i] = i**digitPerLong[i]). Both arrays have
4250   * nonsense values in their 0 and 1 elements, as radices 0 and 1 are not
4251   * used.
4252   */
4253  private static int digitsPerLong[] = {0, 0,
4254      62, 39, 31, 27, 24, 22, 20, 19, 18, 18, 17, 17, 16, 16, 15, 15, 15, 14,
4255      14, 14, 14, 13, 13, 13, 13, 13, 13, 12, 12, 12, 12, 12, 12, 12, 12};
4256
4257  private static BigInteger longRadix[] = {null, null,
4258      valueOf(0x4000000000000000L), valueOf(0x383d9170b85ff80bL),
4259      valueOf(0x4000000000000000L), valueOf(0x6765c793fa10079dL),
4260      valueOf(0x41c21cb8e1000000L), valueOf(0x3642798750226111L),
4261      valueOf(0x1000000000000000L), valueOf(0x12bf307ae81ffd59L),
4262      valueOf(0xde0b6b3a7640000L), valueOf(0x4d28cb56c33fa539L),
4263      valueOf(0x1eca170c00000000L), valueOf(0x780c7372621bd74dL),
4264      valueOf(0x1e39a5057d810000L), valueOf(0x5b27ac993df97701L),

```

```

4265     valueOf(0x1000000000000000L), valueOf(0x27b95e997e21d9f1L),
4266     valueOf(0x5da0e1e53c5c8000L), valueOf( 0xb16a458ef403f19L),
4267     valueOf(0x16bcc41e90000000L), valueOf(0x2d04b7fdd9c0ef49L),
4268     valueOf(0x5658597bcaa24000L), valueOf( 0x6feb266931a75b7L),
4269     valueOf( 0xc29e98000000000L), valueOf(0x14adf4b7320334b9L),
4270     valueOf(0x226ed36478bfa000L), valueOf(0x383d9170b85ff80bL),
4271     valueOf(0x5a3c23e39c000000L), valueOf( 0x4e900abb53e6b71L),
4272     valueOf( 0x7600ec618141000L), valueOf( 0xaee5720ee830681L),
4273     valueOf(0x1000000000000000L), valueOf(0x172588ad4f5f0981L),
4274     valueOf(0x211e44f7d02c1000L), valueOf(0x2ee56725f06e5c71L),
4275     valueOf(0x41c21cb8e1000000L)};
4276
4277     /*
4278     * These two arrays are the integer analogue of above.
4279     */
4280     private static int digitsPerInt[] = {0, 0, 30, 19, 15, 13, 11,
4281     11, 10, 9, 9, 8, 8, 8, 8, 7, 7, 7, 7, 7, 7, 7, 6, 6, 6, 6,
4282     6, 6, 6, 6, 6, 6, 6, 6, 6, 5};
4283
4284     private static int intRadix[] = {0, 0,
4285     0x40000000, 0x4546b3db, 0x40000000, 0x48c27395, 0x159fd800,
4286     0x75db9c97, 0x40000000, 0x17179149, 0x3b9aca00, 0xcc6db61,
4287     0x19a10000, 0x309f1021, 0x57f6c100, 0xa2f1b6f, 0x10000000,
4288     0x18754571, 0x247dbc80, 0x3547667b, 0x4c4b4000, 0x6b5a6e1d,
4289     0x6c20a40, 0x8d2d931, 0xb640000, 0xe8d4a51, 0x1269ae40,
4290     0x17179149, 0x1cb91000, 0x23744899, 0x2b73a840, 0x34e63b41,
4291     0x40000000, 0x4cfa3cc1, 0x5c13d840, 0x6d91b519, 0x39aa400
4292 };
4293
4294     /**
4295     * These routines provide access to the two's complement representation
4296     * of BigIntegers.
4297     */
4298
4299     /**
4300     * Returns the length of the two's complement representation in ints,
4301     * including space for at least one sign bit.
4302     */
4303     private int intLength() {
4304         return (bitLength() >>> 5) + 1;
4305     }
4306
4307     /* Returns sign bit */
4308     private int signBit() {
4309         return signum < 0 ? 1 : 0;
4310     }
4311
4312     /* Returns an int of sign bits */
4313     private int signInt() {
4314         return signum < 0 ? -1 : 0;
4315     }
4316
4317     /**

```

```

4318     * Returns the specified int of the little-endian two's complement
4319     * representation (int 0 is the least significant). The int number can
4320     * be arbitrarily high (values are logically preceded by infinitely many
4321     * sign ints).
4322     */
4323     private int getInt(int n) {
4324         if (n < 0)
4325             return 0;
4326         if (n >= mag.length)
4327             return signInt();
4328
4329         int magInt = mag[mag.length-n-1];
4330
4331         return (signum >= 0 ? magInt :
4332             (n <= firstNonzeroIntNum() ? -magInt : ~magInt));
4333     }
4334
4335     /**
4336     * Returns the index of the int that contains the first nonzero int in the
4337     * little-endian binary representation of the magnitude (int 0 is the
4338     * least significant). If the magnitude is zero, return value is undefined.
4339     */
4340     private int firstNonzeroIntNum() {
4341         int fn = firstNonzeroIntNum - 2;
4342         if (fn == -2) { // firstNonzeroIntNum not initialized yet
4343             fn = 0;
4344
4345             // Search for the first nonzero int
4346             int i;
4347             int mlen = mag.length;
4348             for (i = mlen - 1; i >= 0 && mag[i] == 0; i--)
4349                 ;
4350             fn = mlen - i - 1;
4351             firstNonzeroIntNum = fn + 2; // offset by two to initialize
4352         }
4353         return fn;
4354     }
4355
4356     /** use serialVersionUID from JDK 1.1. for interoperability */
4357     private static final long serialVersionUID = -8287574255936472291L;
4358
4359     /**
4360     * Serializable fields for BigInteger.
4361     *
4362     * @serialField signum int
4363     *         signum of this BigInteger.
4364     * @serialField magnitude int[]
4365     *         magnitude array of this BigInteger.
4366     * @serialField bitCount int
4367     *         number of bits in this BigInteger
4368     * @serialField bitLength int
4369     *         the number of bits in the minimal two's-complement
4370     *         representation of this BigInteger

```

```

4371     * @serialField lowestSetBit int
4372     *         lowest set bit in the twos complement representation
4373     */
4374     private static final ObjectStreamField[] serialPersistentFields = {
4375         new ObjectStreamField("signum", Integer.TYPE),
4376         new ObjectStreamField("magnitude", byte[].class),
4377         new ObjectStreamField("bitCount", Integer.TYPE),
4378         new ObjectStreamField("bitLength", Integer.TYPE),
4379         new ObjectStreamField("firstNonzeroByteNum", Integer.TYPE),
4380         new ObjectStreamField("lowestSetBit", Integer.TYPE)
4381     };
4382
4383     /**
4384     * Reconstitute the {@code BigInteger} instance from a stream (that is,
4385     * deserialize it). The magnitude is read in as an array of bytes
4386     * for historical reasons, but it is converted to an array of ints
4387     * and the byte array is discarded.
4388     * Note:
4389     * The current convention is to initialize the cache fields, bitCount,
4390     * bitLength and lowestSetBit, to 0 rather than some other marker value.
4391     * Therefore, no explicit action to set these fields needs to be taken in
4392     * readObject because those fields already have a 0 value by default since
4393     * defaultReadObject is not being used.
4394     */
4395     private void readObject(java.io.ObjectInputStream s)
4396         throws java.io.IOException, ClassNotFoundException {
4397         /*
4398         * In order to maintain compatibility with previous serialized forms,
4399         * the magnitude of a BigInteger is serialized as an array of bytes.
4400         * The magnitude field is used as a temporary store for the byte array
4401         * that is deserialized. The cached computation fields should be
4402         * transient but are serialized for compatibility reasons.
4403         */
4404
4405         // prepare to read the alternate persistent fields
4406         ObjectInputField fields = s.readFields();
4407
4408         // Read the alternate persistent fields that we care about
4409         int sign = fields.get("signum", -2);
4410         byte[] magnitude = (byte[])fields.get("magnitude", null);
4411
4412         // Validate signum
4413         if (sign < -1 || sign > 1) {
4414             String message = "BigInteger: Invalid signum value";
4415             if (fields.defaulted("signum"))
4416                 message = "BigInteger: Signum not present in stream";
4417             throw new java.io.StreamCorruptedException(message);
4418         }
4419         int[] mag = stripLeadingZeroBytes(magnitude);
4420         if ((mag.length == 0) != (sign == 0)) {
4421             String message = "BigInteger: signum-magnitude mismatch";
4422             if (fields.defaulted("magnitude"))
4423                 message = "BigInteger: Magnitude not present in stream";

```



```

4424         throw new java.io.StreamCorruptedException(message);
4425     }
4426
4427     // Commit final fields via Unsafe
4428     UnsafeHolder.putSign(this, sign);
4429
4430     // Calculate mag field from magnitude and discard magnitude
4431     UnsafeHolder.putMag(this, mag);
4432     if (mag.length >= MAX_MAG_LENGTH) {
4433         try {
4434             checkRange();
4435         } catch (ArithmeticException e) {
4436             throw new java.io.StreamCorruptedException("BigInteger: Out of the supported range"
4437                 );
4438         }
4439     }
4440
4441     // Support for resetting final fields while deserializing
4442     private static class UnsafeHolder {
4443         private static final sun.misc.Unsafe unsafe;
4444         private static final long signumOffset;
4445         private static final long magOffset;
4446         static {
4447             try {
4448                 unsafe = sun.misc.Unsafe.getUnsafe();
4449                 signumOffset = unsafe.objectFieldOffset
4450                     (BigInteger.class.getDeclaredField("signum"));
4451                 magOffset = unsafe.objectFieldOffset
4452                     (BigInteger.class.getDeclaredField("mag"));
4453             } catch (Exception ex) {
4454                 throw new ExceptionInInitializerError(ex);
4455             }
4456         }
4457
4458         static void putSign(BigInteger bi, int sign) {
4459             unsafe.putIntVolatile(bi, signumOffset, sign);
4460         }
4461
4462         static void putMag(BigInteger bi, int[] magnitude) {
4463             unsafe.putObjectVolatile(bi, magOffset, magnitude);
4464         }
4465     }
4466
4467     /**
4468     * Save the {@code BigInteger} instance to a stream.
4469     * The magnitude of a BigInteger is serialized as a byte array for
4470     * historical reasons.
4471     *
4472     * @serialData two necessary fields are written as well as obsolete
4473     *             fields for compatibility with older versions.
4474     */
4475     private void writeObject(ObjectOutputStream s) throws IOException {

```

```

4476     // set the values of the Serializable fields
4477     ObjectOutputStream.PutField fields = s.putFields();
4478     fields.put("signum", signum);
4479     fields.put("magnitude", magSerializedForm());
4480     // The values written for cached fields are compatible with older
4481     // versions, but are ignored in readObject so don't otherwise matter.
4482     fields.put("bitCount", -1);
4483     fields.put("bitLength", -1);
4484     fields.put("lowestSetBit", -2);
4485     fields.put("firstNonzeroByteNum", -2);
4486
4487     // save them
4488     s.writeFields();
4489 }
4490
4491 /**
4492  * Returns the mag array as an array of bytes.
4493  */
4494 private byte[] magSerializedForm() {
4495     int len = mag.length;
4496
4497     int bitLen = (len == 0 ? 0 : ((len - 1) << 5) + bitLengthForInt(mag[0]));
4498     int byteLen = (bitLen + 7) >>> 3;
4499     byte[] result = new byte[byteLen];
4500
4501     for (int i = byteLen - 1, bytesCopied = 4, intIndex = len - 1, nextInt = 0;
4502          i >= 0; i--) {
4503         if (bytesCopied == 4) {
4504             nextInt = mag[intIndex--];
4505             bytesCopied = 1;
4506         } else {
4507             nextInt >>= 8;
4508             bytesCopied++;
4509         }
4510         result[i] = (byte)nextInt;
4511     }
4512     return result;
4513 }
4514
4515 /**
4516  * Converts this {@code BigInteger} to a {@code long}, checking
4517  * for lost information. If the value of this {@code BigInteger}
4518  * is out of the range of the {@code long} type, then an
4519  * {@code ArithmeticException} is thrown.
4520  *
4521  * @return this {@code BigInteger} converted to a {@code long}.
4522  * @throws ArithmeticException if the value of {@code this} will
4523  * not exactly fit in a {@code long}.
4524  * @see BigInteger#longValue
4525  * @since 1.8
4526  */
4527 public long longValueExact() {
4528     if (mag.length <= 2 && bitLength() <= 63)

```

```
4529         return longValue();
4530     else
4531         throw new ArithmeticException("BigInteger out of long range");
4532 }
4533
4534 /**
4535  * Converts this {@code BigInteger} to an {@code int}, checking
4536  * for lost information. If the value of this {@code BigInteger}
4537  * is out of the range of the {@code int} type, then an
4538  * {@code ArithmeticException} is thrown.
4539  *
4540  * @return this {@code BigInteger} converted to an {@code int}.
4541  * @throws ArithmeticException if the value of {@code this} will
4542  * not exactly fit in a {@code int}.
4543  * @see BigInteger#intValue
4544  * @since 1.8
4545  */
4546 public int intValueExact() {
4547     if (mag.length <= 1 && bitLength() <= 31)
4548         return intValue();
4549     else
4550         throw new ArithmeticException("BigInteger out of int range");
4551 }
4552
4553 /**
4554  * Converts this {@code BigInteger} to a {@code short}, checking
4555  * for lost information. If the value of this {@code BigInteger}
4556  * is out of the range of the {@code short} type, then an
4557  * {@code ArithmeticException} is thrown.
4558  *
4559  * @return this {@code BigInteger} converted to a {@code short}.
4560  * @throws ArithmeticException if the value of {@code this} will
4561  * not exactly fit in a {@code short}.
4562  * @see BigInteger#shortValue
4563  * @since 1.8
4564  */
4565 public short shortValueExact() {
4566     if (mag.length <= 1 && bitLength() <= 31) {
4567         int value = intValue();
4568         if (value >= Short.MIN_VALUE && value <= Short.MAX_VALUE)
4569             return shortValue();
4570     }
4571     throw new ArithmeticException("BigInteger out of short range");
4572 }
4573
4574 /**
4575  * Converts this {@code BigInteger} to a {@code byte}, checking
4576  * for lost information. If the value of this {@code BigInteger}
4577  * is out of the range of the {@code byte} type, then an
4578  * {@code ArithmeticException} is thrown.
4579  *
4580  * @return this {@code BigInteger} converted to a {@code byte}.
4581  * @throws ArithmeticException if the value of {@code this} will
```

```

4582     * not exactly fit in a {@code byte}.
4583     * @see BigInteger#byteValue
4584     * @since 1.8
4585     */
4586     public byte byteValueExact() {
4587         if (mag.length <= 1 && bitLength() <= 31) {
4588             int value = intValue();
4589             if (value >= Byte.MIN_VALUE && value <= Byte.MAX_VALUE)
4590                 return byteValue();
4591         }
4592         throw new ArithmeticException("BigInteger out of byte range");
4593     }
4594 }

```

1.3 BitSieve.java

Secuencia 4: java/math/BitSieve.java

```

1  /*
2  * Copyright (c) 1999, 2007, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 package java.math;
27
28 /**
29  * A simple bit sieve used for finding prime number candidates. Allows setting
30  * and clearing of bits in a storage array. The size of the sieve is assumed to
31  * be constant to reduce overhead. All the bits of a new bitSieve are zero, and
32  * bits are removed from it by setting them.
33  *
34  * To reduce storage space and increase efficiency, no even numbers are
35  * represented in the sieve (each bit in the sieve represents an odd number).
36  * The relationship between the index of a bit and the number it represents is

```

```
37  * given by
38  * N = offset + (2*index + 1);
39  * Where N is the integer represented by a bit in the sieve, offset is some
40  * even integer offset indicating where the sieve begins, and index is the
41  * index of a bit in the sieve array.
42  *
43  * @see      BigInteger
44  * @author   Michael McCloskey
45  * @since    1.3
46  */
47  class BitSieve {
48      /**
49       * Stores the bits in this bitSieve.
50       */
51      private long bits[];
52
53      /**
54       * Length is how many bits this sieve holds.
55       */
56      private int length;
57
58      /**
59       * A small sieve used to filter out multiples of small primes in a search
60       * sieve.
61       */
62      private static BitSieve smallSieve = new BitSieve();
63
64      /**
65       * Construct a "small sieve" with a base of 0. This constructor is
66       * used internally to generate the set of "small primes" whose multiples
67       * are excluded from sieves generated by the main (package private)
68       * constructor, BitSieve(BigInteger base, int searchLen). The length
69       * of the sieve generated by this constructor was chosen for performance;
70       * it controls a tradeoff between how much time is spent constructing
71       * other sieves, and how much time is wasted testing composite candidates
72       * for primality. The length was chosen experimentally to yield good
73       * performance.
74       */
75      private BitSieve() {
76          length = 150 * 64;
77          bits = new long[(unitIndex(length - 1) + 1)];
78
79          // Mark 1 as composite
80          set(0);
81          int nextIndex = 1;
82          int nextPrime = 3;
83
84          // Find primes and remove their multiples from sieve
85          do {
86              sieveSingle(length, nextIndex + nextPrime, nextPrime);
87              nextIndex = sieveSearch(length, nextIndex + 1);
88              nextPrime = 2*nextIndex + 1;
89          } while((nextIndex > 0) && (nextPrime < length));
```

```

90     }
91
92     /**
93      * Construct a bit sieve of searchLen bits used for finding prime number
94      * candidates. The new sieve begins at the specified base, which must
95      * be even.
96      */
97     BitSieve(BigInteger base, int searchLen) {
98         /**
99          * Candidates are indicated by clear bits in the sieve. As a candidates
100          * nonprimality is calculated, a bit is set in the sieve to eliminate
101          * it. To reduce storage space and increase efficiency, no even numbers
102          * are represented in the sieve (each bit in the sieve represents an
103          * odd number).
104          */
105         bits = new long[(unitIndex(searchLen-1) + 1)];
106         length = searchLen;
107         int start = 0;
108
109         int step = smallSieve.sieveSearch(smallSieve.length, start);
110         int convertedStep = (step * 2) + 1;
111
112         // Construct the large sieve at an even offset specified by base
113         MutableBigInteger b = new MutableBigInteger(base);
114         MutableBigInteger q = new MutableBigInteger();
115         do {
116             // Calculate base mod convertedStep
117             start = b.divideOneWord(convertedStep, q);
118
119             // Take each multiple of step out of sieve
120             start = convertedStep - start;
121             if (start % 2 == 0)
122                 start += convertedStep;
123             sieveSingle(searchLen, (start-1)/2, convertedStep);
124
125             // Find next prime from small sieve
126             step = smallSieve.sieveSearch(smallSieve.length, step+1);
127             convertedStep = (step * 2) + 1;
128         } while (step > 0);
129     }
130
131     /**
132      * Given a bit index return unit index containing it.
133      */
134     private static int unitIndex(int bitIndex) {
135         return bitIndex >>> 6;
136     }
137
138     /**
139      * Return a unit that masks the specified bit in its unit.
140      */
141     private static long bit(int bitIndex) {
142         return 1L << (bitIndex & ((1<<6) - 1));

```

```
143     }
144
145     /**
146      * Get the value of the bit at the specified index.
147      */
148     private boolean get(int bitIndex) {
149         int unitIndex = unitIndex(bitIndex);
150         return ((bits[unitIndex] & bit(bitIndex)) != 0);
151     }
152
153     /**
154      * Set the bit at the specified index.
155      */
156     private void set(int bitIndex) {
157         int unitIndex = unitIndex(bitIndex);
158         bits[unitIndex] |= bit(bitIndex);
159     }
160
161     /**
162      * This method returns the index of the first clear bit in the search
163      * array that occurs at or after start. It will not search past the
164      * specified limit. It returns -1 if there is no such clear bit.
165      */
166     private int sieveSearch(int limit, int start) {
167         if (start >= limit)
168             return -1;
169
170         int index = start;
171         do {
172             if (!get(index))
173                 return index;
174             index++;
175         } while(index < limit-1);
176         return -1;
177     }
178
179     /**
180      * Sieve a single set of multiples out of the sieve. Begin to remove
181      * multiples of the specified step starting at the specified start index,
182      * up to the specified limit.
183      */
184     private void sieveSingle(int limit, int start, int step) {
185         while(start < limit) {
186             set(start);
187             start += step;
188         }
189     }
190
191     /**
192      * Test probable primes in the sieve and return successful candidates.
193      */
194     BigInteger retrieve(BigInteger initValue, int certainty, java.util.Random random) {
195         // Examine the sieve one long at a time to find possible primes
```

```

196     int offset = 1;
197     for (int i=0; i<bits.length; i++) {
198         long nextLong = ~bits[i];
199         for (int j=0; j<64; j++) {
200             if ((nextLong & 1) == 1) {
201                 BigInteger candidate = initValue.add(
202                     BigInteger.valueOf(offset));
203                 if (candidate.primeToCertainty(certainty, random))
204                     return candidate;
205             }
206             nextLong >>= 1;
207             offset+=2;
208         }
209     }
210     return null;
211 }
212 }

```

1.4 MathContext.java

Secuencia 5: java/math/MathContext.java

```

1  /*
2  * Copyright (c) 2003, 2007, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * Portions Copyright IBM Corporation, 1997, 2001. All Rights Reserved.
28 */
29
30 package java.math;
31 import java.io.*;
32

```



```

33 /**
34  * Immutable objects which encapsulate the context settings which
35  * describe certain rules for numerical operators, such as those
36  * implemented by the {@link BigDecimal} class.
37  *
38  * <p>The base-independent settings are:
39  * <ol>
40  * <li>{@code precision}:
41  * the number of digits to be used for an operation; results are
42  * rounded to this precision
43  *
44  * <li>{@code roundingMode}:
45  * a {@link RoundingMode} object which specifies the algorithm to be
46  * used for rounding.
47  * </ol>
48  *
49  * @see      BigDecimal
50  * @see      RoundingMode
51  * @author    Mike Cowlishaw
52  * @author    Joseph D. Darcy
53  * @since 1.5
54  */
55
56 public final class MathContext implements Serializable {
57
58     /* ----- Constants ----- */
59
60     // defaults for constructors
61     private static final int DEFAULT_DIGITS = 9;
62     private static final RoundingMode DEFAULT_ROUNDINGMODE = RoundingMode.HALF_UP;
63     // Smallest values for digits (Maximum is Integer.MAX_VALUE)
64     private static final int MIN_DIGITS = 0;
65
66     // Serialization version
67     private static final long serialVersionUID = 5579720004786848255L;
68
69     /* ----- Public Properties ----- */
70     /**
71      * A {@code MathContext} object whose settings have the values
72      * required for unlimited precision arithmetic.
73      * The values of the settings are:
74      * <code>
75      * precision=0 roundingMode=HALF_UP
76      * </code>
77      */
78     public static final MathContext UNLIMITED =
79         new MathContext(0, RoundingMode.HALF_UP);
80
81     /**
82      * A {@code MathContext} object with a precision setting
83      * matching the IEEE 754R Decimal32 format, 7 digits, and a
84      * rounding mode of {@link RoundingMode#HALF_EVEN HALF_EVEN}, the
85      * IEEE 754R default.

```

```

86     */
87     public static final MathContext DECIMAL32 =
88         new MathContext(7, RoundingMode.HALF_EVEN);
89
90     /**
91     * A {@code MathContext} object with a precision setting
92     * matching the IEEE 754R Decimal64 format, 16 digits, and a
93     * rounding mode of {@link RoundingMode#HALF_EVEN HALF_EVEN}, the
94     * IEEE 754R default.
95     */
96     public static final MathContext DECIMAL64 =
97         new MathContext(16, RoundingMode.HALF_EVEN);
98
99     /**
100    * A {@code MathContext} object with a precision setting
101    * matching the IEEE 754R Decimal128 format, 34 digits, and a
102    * rounding mode of {@link RoundingMode#HALF_EVEN HALF_EVEN}, the
103    * IEEE 754R default.
104    */
105    public static final MathContext DECIMAL128 =
106        new MathContext(34, RoundingMode.HALF_EVEN);
107
108    /* ----- Shared Properties ----- */
109    /**
110    * The number of digits to be used for an operation. A value of 0
111    * indicates that unlimited precision (as many digits as are
112    * required) will be used. Note that leading zeros (in the
113    * coefficient of a number) are never significant.
114    *
115    * <p>{@code precision} will always be non-negative.
116    *
117    * @serial
118    */
119    final int precision;
120
121    /**
122    * The rounding algorithm to be used for an operation.
123    *
124    * @see RoundingMode
125    * @serial
126    */
127    final RoundingMode roundingMode;
128
129    /* ----- Constructors ----- */
130
131    /**
132    * Constructs a new {@code MathContext} with the specified
133    * precision and the {@link RoundingMode#HALF_UP HALF_UP} rounding
134    * mode.
135    *
136    * @param setPrecision The non-negative {@code int} precision setting.
137    * @throws IllegalArgumentException if the {@code setPrecision} parameter is less
138    *         than zero.

```

```

139     */
140     public MathContext(int setPrecision) {
141         this(setPrecision, DEFAULT_ROUNDINGMODE);
142         return;
143     }
144
145     /**
146      * Constructs a new {@code MathContext} with a specified
147      * precision and rounding mode.
148      *
149      * @param setPrecision The non-negative {@code int} precision setting.
150      * @param setRoundingMode The rounding mode to use.
151      * @throws IllegalArgumentException if the {@code setPrecision} parameter is less
152      *         than zero.
153      * @throws NullPointerException if the rounding mode argument is {@code null}
154      */
155     public MathContext(int setPrecision,
156                       RoundingMode setRoundingMode) {
157         if (setPrecision < MIN_DIGITS)
158             throw new IllegalArgumentException("Digits < 0");
159         if (setRoundingMode == null)
160             throw new NullPointerException("null RoundingMode");
161
162         precision = setPrecision;
163         roundingMode = setRoundingMode;
164         return;
165     }
166
167     /**
168      * Constructs a new {@code MathContext} from a string.
169      *
170      * The string must be in the same format as that produced by the
171      * {@link #toString} method.
172      *
173      * <p>An {@code IllegalArgumentException} is thrown if the precision
174      * section of the string is out of range ({@code < 0}) or the string is
175      * not in the format created by the {@link #toString} method.
176      *
177      * @param val The string to be parsed
178      * @throws IllegalArgumentException if the precision section is out of range
179      *         or of incorrect format
180      * @throws NullPointerException if the argument is {@code null}
181      */
182     public MathContext(String val) {
183         boolean bad = false;
184         int setPrecision;
185         if (val == null)
186             throw new NullPointerException("null String");
187         try { // any error here is a string format problem
188             if (!val.startsWith("precision=")) throw new RuntimeException();
189             int fence = val.indexOf(' '); // could be -1
190             int off = 10; // where value starts
191             setPrecision = Integer.parseInt(val.substring(10, fence));

```

```

192         if (!val.startsWith("roundingMode=", fence+1))
193             throw new RuntimeException();
194         off = fence + 1 + 13;
195         String str = val.substring(off, val.length());
196         roundingMode = RoundingMode.valueOf(str);
197     } catch (RuntimeException re) {
198         throw new IllegalArgumentException("bad string format");
199     }
200 }
201
202 if (setPrecision < MIN_DIGITS)
203     throw new IllegalArgumentException("Digits < 0");
204 // the other parameters cannot be invalid if we got here
205 precision = setPrecision;
206 }
207
208 /**
209  * Returns the {@code precision} setting.
210  * This value is always non-negative.
211  *
212  * @return an {@code int} which is the value of the {@code precision}
213  *         setting
214  */
215 public int getPrecision() {
216     return precision;
217 }
218
219 /**
220  * Returns the roundingMode setting.
221  * This will be one of
222  * {@link RoundingMode#CEILING},
223  * {@link RoundingMode#DOWN},
224  * {@link RoundingMode#FLOOR},
225  * {@link RoundingMode#HALF_DOWN},
226  * {@link RoundingMode#HALF_EVEN},
227  * {@link RoundingMode#HALF_UP},
228  * {@link RoundingMode#UNNECESSARY}, or
229  * {@link RoundingMode#UP}.
230  *
231  * @return a {@code RoundingMode} object which is the value of the
232  *         {@code roundingMode} setting
233  */
234
235 public RoundingMode getRoundingMode() {
236     return roundingMode;
237 }
238
239 /**
240  * Compares this {@code MathContext} with the specified
241  * {@code Object} for equality.
242  *
243  * @param x {@code Object} to which this {@code MathContext} is to
244  *         be compared.

```

```

245     * @return {@code true} if and only if the specified {@code Object} is
246     *         a {@code MathContext} object which has exactly the same
247     *         settings as this object
248     */
249     public boolean equals(Object x){
250         MathContext mc;
251         if (!(x instanceof MathContext))
252             return false;
253         mc = (MathContext) x;
254         return mc.precision == this.precision
255             && mc.roundingMode == this.roundingMode; // no need for .equals()
256     }
257
258     /**
259     * Returns the hash code for this {@code MathContext}.
260     *
261     * @return hash code for this {@code MathContext}
262     */
263     public int hashCode() {
264         return this.precision + roundingMode.hashCode() * 59;
265     }
266
267     /**
268     * Returns the string representation of this {@code MathContext}.
269     * The {@code String} returned represents the settings of the
270     * {@code MathContext} object as two space-delimited words
271     * (separated by a single space character, <tt>'&#92;u0020'</tt>,
272     * and with no leading or trailing white space), as follows:
273     * <ol>
274     * <li>
275     * The string {@code "precision="}, immediately followed
276     * by the value of the precision setting as a numeric string as if
277     * generated by the {@link Integer#toString(int) Integer.toString}
278     * method.
279     *
280     * <li>
281     * The string {@code "roundingMode="}, immediately
282     * followed by the value of the {@code roundingMode} setting as a
283     * word. This word will be the same as the name of the
284     * corresponding public constant in the {@link RoundingMode}
285     * enum.
286     * </ol>
287     * <p>
288     * For example:
289     * <pre>
290     * precision=9 roundingMode=HALF_UP
291     * </pre>
292     *
293     * Additional words may be appended to the result of
294     * {@code toString} in the future if more properties are added to
295     * this class.
296     *
297     * @return a {@code String} representing the context settings

```

```

298     */
299     public java.lang.String toString() {
300         return "precision=" + precision + " " +
301             "roundingMode=" + roundingMode.toString();
302     }
303
304     // Private methods
305
306     /**
307      * Reconstitute the {@code MathContext} instance from a stream (that is,
308      * deserialize it).
309      *
310      * @param s the stream being read.
311      */
312     private void readObject(java.io.ObjectInputStream s)
313         throws java.io.IOException, ClassNotFoundException {
314         s.defaultReadObject();    // read in all fields
315         // validate possibly bad fields
316         if (precision < MIN_DIGITS) {
317             String message = "MathContext: invalid digits in stream";
318             throw new java.io.StreamCorruptedException(message);
319         }
320         if (roundingMode == null) {
321             String message = "MathContext: null roundingMode in stream";
322             throw new java.io.StreamCorruptedException(message);
323         }
324     }
325
326 }

```

1.5 MutableBigInteger.java

Secuencia 6: java/math/MutableBigInteger.java

```

1  /*
2  * Copyright (c) 1999, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *

```

```
21  *
22  *
23  *
24  */
25
26  package java.math;
27
28  /**
29   * A class used to represent multiprecision integers that makes efficient
30   * use of allocated space by allowing a number to occupy only part of
31   * an array so that the arrays do not have to be reallocated as often.
32   * When performing an operation with many iterations the array used to
33   * hold a number is only reallocated when necessary and does not have to
34   * be the same size as the number it represents. A mutable number allows
35   * calculations to occur on the same number without having to create
36   * a new number for every step of the calculation as occurs with
37   * BigIntegers.
38   *
39   * @see      BigInteger
40   * @author    Michael McCloskey
41   * @author    Timothy Bukt
42   * @since     1.3
43   */
44
45  import static java.math.BigDecimal.INFLATED;
46  import static java.math.BigInteger.LONG_MASK;
47  import java.util.Arrays;
48
49  class MutableBigInteger {
50      /**
51       * Holds the magnitude of this MutableBigInteger in big endian order.
52       * The magnitude may start at an offset into the value array, and it may
53       * end before the length of the value array.
54       */
55      int[] value;
56
57      /**
58       * The number of ints of the value array that are currently used
59       * to hold the magnitude of this MutableBigInteger. The magnitude starts
60       * at an offset and offset + intLen may be less than value.length.
61       */
62      int intLen;
63
64      /**
65       * The offset into the value array where the magnitude of this
66       * MutableBigInteger begins.
67       */
68      int offset = 0;
69
70      // Constants
71      /**
72       * MutableBigInteger with one element value array with the value 1. Used by
73       * BigDecimal divideAndRound to increment the quotient. Use this constant
```

```
74     * only when the method is not going to modify this object.
75     */
76     static final MutableBigInteger ONE = new MutableBigInteger(1);
77
78     /**
79     * The minimum {@code intLen} for cancelling powers of two before
80     * dividing.
81     * If the number of ints is less than this threshold,
82     * {@code divideKnuth} does not eliminate common powers of two from
83     * the dividend and divisor.
84     */
85     static final int KNUTH_POW2_THRESH_LEN = 6;
86
87     /**
88     * The minimum number of trailing zero ints for cancelling powers of two
89     * before dividing.
90     * If the dividend and divisor don't share at least this many zero ints
91     * at the end, {@code divideKnuth} does not eliminate common powers
92     * of two from the dividend and divisor.
93     */
94     static final int KNUTH_POW2_THRESH_ZEROS = 3;
95
96     // Constructors
97
98     /**
99     * The default constructor. An empty MutableBigInteger is created with
100    * a one word capacity.
101    */
102    MutableBigInteger() {
103        value = new int[1];
104        intLen = 0;
105    }
106
107    /**
108    * Construct a new MutableBigInteger with a magnitude specified by
109    * the int val.
110    */
111    MutableBigInteger(int val) {
112        value = new int[1];
113        intLen = 1;
114        value[0] = val;
115    }
116
117    /**
118    * Construct a new MutableBigInteger with the specified value array
119    * up to the length of the array supplied.
120    */
121    MutableBigInteger(int[] val) {
122        value = val;
123        intLen = val.length;
124    }
125
126    /**
```



```

127     * Construct a new MutableBigInteger with a magnitude equal to the
128     * specified BigInteger.
129     */
130     MutableBigInteger(BigInteger b) {
131         intLen = b.mag.length;
132         value = Arrays.copyOf(b.mag, intLen);
133     }
134
135     /**
136     * Construct a new MutableBigInteger with a magnitude equal to the
137     * specified MutableBigInteger.
138     */
139     MutableBigInteger(MutableBigInteger val) {
140         intLen = val.intLen;
141         value = Arrays.copyOfRange(val.value, val.offset, val.offset + intLen);
142     }
143
144     /**
145     * Makes this number an {@code n}-int number all of whose bits are ones.
146     * Used by Burnikel-Ziegler division.
147     * @param n number of ints in the {@code value} array
148     * @return a number equal to {@code ((1<<(32*n)))-1}
149     */
150     private void ones(int n) {
151         if (n > value.length)
152             value = new int[n];
153         Arrays.fill(value, -1);
154         offset = 0;
155         intLen = n;
156     }
157
158     /**
159     * Internal helper method to return the magnitude array. The caller is not
160     * supposed to modify the returned array.
161     */
162     private int[] getMagnitudeArray() {
163         if (offset > 0 || value.length != intLen)
164             return Arrays.copyOfRange(value, offset, offset + intLen);
165         return value;
166     }
167
168     /**
169     * Convert this MutableBigInteger to a long value. The caller has to make
170     * sure this MutableBigInteger can be fit into long.
171     */
172     private long toLong() {
173         assert (intLen <= 2) : "this MutableBigInteger exceeds the range of long";
174         if (intLen == 0)
175             return 0;
176         long d = value[offset] & LONG_MASK;
177         return (intLen == 2) ? d << 32 | (value[offset + 1] & LONG_MASK) : d;
178     }
179

```

```

180  /**
181   * Convert this MutableBigInteger to a BigInteger object.
182   */
183  BigInteger toBigInteger(int sign) {
184      if (intLen == 0 || sign == 0)
185          return BigInteger.ZERO;
186      return new BigInteger(getMagnitudeArray(), sign);
187  }
188
189  /**
190   * Converts this number to a nonnegative {@code BigInteger}.
191   */
192  BigInteger toBigInteger() {
193      normalize();
194      return toBigInteger(isZero() ? 0 : 1);
195  }
196
197  /**
198   * Convert this MutableBigInteger to BigDecimal object with the specified sign
199   * and scale.
200   */
201  BigDecimal toBigDecimal(int sign, int scale) {
202      if (intLen == 0 || sign == 0)
203          return BigDecimal.valueOf(scale);
204      int[] mag = getMagnitudeArray();
205      int len = mag.length;
206      int d = mag[0];
207      // If this MutableBigInteger can't be fit into long, we need to
208      // make a BigInteger object for the resultant BigDecimal object.
209      if (len > 2 || (d < 0 && len == 2))
210          return new BigDecimal(new BigInteger(mag, sign), INFLATED, scale, 0);
211      long v = (len == 2) ?
212          ((mag[1] & LONG_MASK) | (d & LONG_MASK) << 32) :
213          d & LONG_MASK;
214      return BigDecimal.valueOf(sign == -1 ? -v : v, scale);
215  }
216
217  /**
218   * This is for internal use in converting from a MutableBigInteger
219   * object into a long value given a specified sign.
220   * returns INFLATED if value is not fit into long
221   */
222  long toCompactValue(int sign) {
223      if (intLen == 0 || sign == 0)
224          return 0L;
225      int[] mag = getMagnitudeArray();
226      int len = mag.length;
227      int d = mag[0];
228      // If this MutableBigInteger can not be fitted into long, we need to
229      // make a BigInteger object for the resultant BigDecimal object.
230      if (len > 2 || (d < 0 && len == 2))
231          return INFLATED;
232      long v = (len == 2) ?

```

```

233         ((mag[1] & LONG_MASK) | (d & LONG_MASK) << 32) :
234         d & LONG_MASK;
235     return sign == -1 ? -v : v;
236 }
237
238 /**
239  * Clear out a MutableBigInteger for reuse.
240  */
241 void clear() {
242     offset = intLen = 0;
243     for (int index=0, n=value.length; index < n; index++)
244         value[index] = 0;
245 }
246
247 /**
248  * Set a MutableBigInteger to zero, removing its offset.
249  */
250 void reset() {
251     offset = intLen = 0;
252 }
253
254 /**
255  * Compare the magnitude of two MutableBigIntegers. Returns -1, 0 or 1
256  * as this MutableBigInteger is numerically less than, equal to, or
257  * greater than <tt>b</tt>.
258  */
259 final int compare(MutableBigInteger b) {
260     int blen = b.intLen;
261     if (intLen < blen)
262         return -1;
263     if (intLen > blen)
264         return 1;
265
266     // Add Integer.MIN_VALUE to make the comparison act as unsigned integer
267     // comparison.
268     int[] bval = b.value;
269     for (int i = offset, j = b.offset; i < intLen + offset; i++, j++) {
270         int b1 = value[i] + 0x80000000;
271         int b2 = bval[j] + 0x80000000;
272         if (b1 < b2)
273             return -1;
274         if (b1 > b2)
275             return 1;
276     }
277     return 0;
278 }
279
280 /**
281  * Returns a value equal to what {@code b.leftShift(32*ints); return compare(b);}
282  * would return, but doesn't change the value of {@code b}.
283  */
284 private int compareShifted(MutableBigInteger b, int ints) {
285     int blen = b.intLen;

```

```

286     int alen = intLen - ints;
287     if (alen < blen)
288         return -1;
289     if (alen > blen)
290         return 1;
291
292     // Add Integer.MIN_VALUE to make the comparison act as unsigned integer
293     // comparison.
294     int[] bval = b.value;
295     for (int i = offset, j = b.offset; i < alen + offset; i++, j++) {
296         int b1 = value[i] + 0x80000000;
297         int b2 = bval[j] + 0x80000000;
298         if (b1 < b2)
299             return -1;
300         if (b1 > b2)
301             return 1;
302     }
303     return 0;
304 }
305
306 /**
307  * Compare this against half of a MutableBigInteger object (Needed for
308  * remainder tests).
309  * Assumes no leading unnecessary zeros, which holds for results
310  * from divide().
311  */
312 final int compareHalf(MutableBigInteger b) {
313     int blen = b.intLen;
314     int len = intLen;
315     if (len <= 0)
316         return blen <= 0 ? 0 : -1;
317     if (len > blen)
318         return 1;
319     if (len < blen - 1)
320         return -1;
321     int[] bval = b.value;
322     int bstart = 0;
323     int carry = 0;
324     // Only 2 cases left: len == blen or len == blen - 1
325     if (len != blen) { // len == blen - 1
326         if (bval[bstart] == 1) {
327             ++bstart;
328             carry = 0x80000000;
329         } else
330             return -1;
331     }
332     // compare values with right-shifted values of b,
333     // carrying shifted-out bits across words
334     int[] val = value;
335     for (int i = offset, j = bstart; i < len + offset; i++) {
336         int bv = bval[j++];
337         long hb = ((bv >>> 1) + carry) & LONG_MASK;
338         long v = val[i++] & LONG_MASK;

```

```

339         if (v != hb)
340             return v < hb ? -1 : 1;
341         carry = (bv & 1) << 31; // carry will be either 0x80000000 or 0
342     }
343     return carry == 0 ? 0 : -1;
344 }
345
346 /**
347  * Return the index of the lowest set bit in this MutableBigInteger. If the
348  * magnitude of this MutableBigInteger is zero, -1 is returned.
349  */
350 private final int getLowestSetBit() {
351     if (intLen == 0)
352         return -1;
353     int j, b;
354     for (j=intLen-1; (j > 0) && (value[j+offset] == 0); j--)
355         ;
356     b = value[j+offset];
357     if (b == 0)
358         return -1;
359     return ((intLen-1-j)<<5) + Integer.numberOfTrailingZeros(b);
360 }
361
362 /**
363  * Return the int in use in this MutableBigInteger at the specified
364  * index. This method is not used because it is not inlined on all
365  * platforms.
366  */
367 private final int getInt(int index) {
368     return value[offset+index];
369 }
370
371 /**
372  * Return a long which is equal to the unsigned value of the int in
373  * use in this MutableBigInteger at the specified index. This method is
374  * not used because it is not inlined on all platforms.
375  */
376 private final long getLong(int index) {
377     return value[offset+index] & LONG_MASK;
378 }
379
380 /**
381  * Ensure that the MutableBigInteger is in normal form, specifically
382  * making sure that there are no leading zeros, and that if the
383  * magnitude is zero, then intLen is zero.
384  */
385 final void normalize() {
386     if (intLen == 0) {
387         offset = 0;
388         return;
389     }
390
391     int index = offset;

```

```
392     if (value[index] != 0)
393         return;
394
395     int indexBound = index+intLen;
396     do {
397         index++;
398     } while(index < indexBound && value[index] == 0);
399
400     int numZeros = index - offset;
401     intLen -= numZeros;
402     offset = (intLen == 0 ? 0 : offset+numZeros);
403 }
404
405 /**
406  * If this MutableBigInteger cannot hold len words, increase the size
407  * of the value array to len words.
408  */
409 private final void ensureCapacity(int len) {
410     if (value.length < len) {
411         value = new int[len];
412         offset = 0;
413         intLen = len;
414     }
415 }
416
417 /**
418  * Convert this MutableBigInteger into an int array with no leading
419  * zeros, of a length that is equal to this MutableBigInteger's intLen.
420  */
421 int[] toIntArray() {
422     int[] result = new int[intLen];
423     for(int i=0; i < intLen; i++)
424         result[i] = value[offset+i];
425     return result;
426 }
427
428 /**
429  * Sets the int at index+offset in this MutableBigInteger to val.
430  * This does not get inlined on all platforms so it is not used
431  * as often as originally intended.
432  */
433 void setInt(int index, int val) {
434     value[offset + index] = val;
435 }
436
437 /**
438  * Sets this MutableBigInteger's value array to the specified array.
439  * The intLen is set to the specified length.
440  */
441 void setValue(int[] val, int length) {
442     value = val;
443     intLen = length;
444     offset = 0;
```

```
445     }
446
447     /**
448     * Sets this MutableBigInteger's value array to a copy of the specified
449     * array. The intLen is set to the length of the new array.
450     */
451     void copyValue(MutableBigInteger src) {
452         int len = src.intLen;
453         if (value.length < len)
454             value = new int[len];
455         System.arraycopy(src.value, src.offset, value, 0, len);
456         intLen = len;
457         offset = 0;
458     }
459
460     /**
461     * Sets this MutableBigInteger's value array to a copy of the specified
462     * array. The intLen is set to the length of the specified array.
463     */
464     void copyValue(int[] val) {
465         int len = val.length;
466         if (value.length < len)
467             value = new int[len];
468         System.arraycopy(val, 0, value, 0, len);
469         intLen = len;
470         offset = 0;
471     }
472
473     /**
474     * Returns true iff this MutableBigInteger has a value of one.
475     */
476     boolean isOne() {
477         return (intLen == 1) && (value[offset] == 1);
478     }
479
480     /**
481     * Returns true iff this MutableBigInteger has a value of zero.
482     */
483     boolean isZero() {
484         return (intLen == 0);
485     }
486
487     /**
488     * Returns true iff this MutableBigInteger is even.
489     */
490     boolean isEven() {
491         return (intLen == 0) || ((value[offset + intLen - 1] & 1) == 0);
492     }
493
494     /**
495     * Returns true iff this MutableBigInteger is odd.
496     */
497     boolean isOdd() {
```

```

498     return isZero() ? false : ((value[offset + intLen - 1] & 1) == 1);
499 }
500
501 /**
502  * Returns true iff this MutableBigInteger is in normal form. A
503  * MutableBigInteger is in normal form if it has no leading zeros
504  * after the offset, and intLen + offset <= value.length.
505  */
506 boolean isNormal() {
507     if (intLen + offset > value.length)
508         return false;
509     if (intLen == 0)
510         return true;
511     return (value[offset] != 0);
512 }
513
514 /**
515  * Returns a String representation of this MutableBigInteger in radix 10.
516  */
517 public String toString() {
518     BigInteger b = toBigInteger(1);
519     return b.toString();
520 }
521
522 /**
523  * Like {@link #rightShift(int)} but {@code n} can be greater than the length of the number.
524  */
525 void safeRightShift(int n) {
526     if (n/32 >= intLen) {
527         reset();
528     } else {
529         rightShift(n);
530     }
531 }
532
533 /**
534  * Right shift this MutableBigInteger n bits. The MutableBigInteger is left
535  * in normal form.
536  */
537 void rightShift(int n) {
538     if (intLen == 0)
539         return;
540     int nInts = n >>> 5;
541     int nBits = n & 0x1F;
542     this.intLen -= nInts;
543     if (nBits == 0)
544         return;
545     int bitsInHighWord = BigInteger.bitLengthForInt(value[offset]);
546     if (nBits >= bitsInHighWord) {
547         this.primitiveLeftShift(32 - nBits);
548         this.intLen--;
549     } else {
550         primitiveRightShift(nBits);

```



```

551     }
552 }
553
554 /**
555  * Like {@link #leftShift(int)} but {@code n} can be zero.
556  */
557 void safeLeftShift(int n) {
558     if (n > 0) {
559         leftShift(n);
560     }
561 }
562
563 /**
564  * Left shift this MutableBigInteger n bits.
565  */
566 void leftShift(int n) {
567     /*
568      * If there is enough storage space in this MutableBigInteger already
569      * the available space will be used. Space to the right of the used
570      * ints in the value array is faster to utilize, so the extra space
571      * will be taken from the right if possible.
572      */
573     if (intLen == 0)
574         return;
575     int nInts = n >>> 5;
576     int nBits = n&0x1F;
577     int bitsInHighWord = BigInteger.bitLengthForInt(value[offset]);
578
579     // If shift can be done without moving words, do so
580     if (n <= (32-bitsInHighWord)) {
581         primitiveLeftShift(nBits);
582         return;
583     }
584
585     int newLen = intLen + nInts +1;
586     if (nBits <= (32-bitsInHighWord))
587         newLen--;
588     if (value.length < newLen) {
589         // The array must grow
590         int[] result = new int[newLen];
591         for (int i=0; i < intLen; i++)
592             result[i] = value[offset+i];
593         setValue(result, newLen);
594     } else if (value.length - offset >= newLen) {
595         // Use space on right
596         for(int i=0; i < newLen - intLen; i++)
597             value[offset+intLen+i] = 0;
598     } else {
599         // Must use space on left
600         for (int i=0; i < intLen; i++)
601             value[i] = value[offset+i];
602         for (int i=intLen; i < newLen; i++)
603             value[i] = 0;

```

```

604         offset = 0;
605     }
606     intLen = newLen;
607     if (nBits == 0)
608         return;
609     if (nBits <= (32-bitsInHighWord))
610         primitiveLeftShift(nBits);
611     else
612         primitiveRightShift(32 -nBits);
613 }
614
615 /**
616  * A primitive used for division. This method adds in one multiple of the
617  * divisor a back to the dividend result at a specified offset. It is used
618  * when qhat was estimated too large, and must be adjusted.
619  */
620 private int divadd(int[] a, int[] result, int offset) {
621     long carry = 0;
622
623     for (int j=a.length-1; j >= 0; j--) {
624         long sum = (a[j] & LONG_MASK) +
625                 (result[j+offset] & LONG_MASK) + carry;
626         result[j+offset] = (int)sum;
627         carry = sum >>> 32;
628     }
629     return (int)carry;
630 }
631
632 /**
633  * This method is used for division. It multiplies an n word input a by one
634  * word input x, and subtracts the n word product from q. This is needed
635  * when subtracting qhat*divisor from dividend.
636  */
637 private int mulsub(int[] q, int[] a, int x, int len, int offset) {
638     long xLong = x & LONG_MASK;
639     long carry = 0;
640     offset += len;
641
642     for (int j=len-1; j >= 0; j--) {
643         long product = (a[j] & LONG_MASK) * xLong + carry;
644         long difference = q[offset] - product;
645         q[offset--] = (int)difference;
646         carry = (product >>> 32)
647                 + (((difference & LONG_MASK) >
648                    (((~(int)product) & LONG_MASK))) ? 1:0);
649     }
650     return (int)carry;
651 }
652
653 /**
654  * The method is the same as mulsum, except the fact that q array is not
655  * updated, the only result of the method is borrow flag.
656  */

```

```

657 private int mulsubBorrow(int[] q, int[] a, int x, int len, int offset) {
658     long xLong = x & LONG_MASK;
659     long carry = 0;
660     offset += len;
661     for (int j=len-1; j >= 0; j--) {
662         long product = (a[j] & LONG_MASK) * xLong + carry;
663         long difference = q[offset--] - product;
664         carry = (product >>> 32)
665             + (((difference & LONG_MASK) >
666                (((~(int)product) & LONG_MASK))) ? 1:0);
667     }
668     return (int)carry;
669 }
670
671 /**
672  * Right shift this MutableBigInteger n bits, where n is
673  * less than 32.
674  * Assumes that intLen > 0, n > 0 for speed
675  */
676 private final void primitiveRightShift(int n) {
677     int[] val = value;
678     int n2 = 32 - n;
679     for (int i=offset+intLen-1, c=val[i]; i > offset; i--) {
680         int b = c;
681         c = val[i-1];
682         val[i] = (c << n2) | (b >>> n);
683     }
684     val[offset] >>>= n;
685 }
686
687 /**
688  * Left shift this MutableBigInteger n bits, where n is
689  * less than 32.
690  * Assumes that intLen > 0, n > 0 for speed
691  */
692 private final void primitiveLeftShift(int n) {
693     int[] val = value;
694     int n2 = 32 - n;
695     for (int i=offset, c=val[i], m=i+intLen-1; i < m; i++) {
696         int b = c;
697         c = val[i+1];
698         val[i] = (b << n) | (c >>> n2);
699     }
700     val[offset+intLen-1] <<= n;
701 }
702
703 /**
704  * Returns a {@code BigInteger} equal to the {@code n}
705  * low ints of this number.
706  */
707 private BigInteger getLower(int n) {
708     if (isZero()) {
709         return BigInteger.ZERO;

```

```

710     } else if (intLen < n) {
711         return toBigInteger(1);
712     } else {
713         // strip zeros
714         int len = n;
715         while (len > 0 && value[offset+intLen-len] == 0)
716             len--;
717         int sign = len > 0 ? 1 : 0;
718         return new BigInteger(Arrays.copyOfRange(value, offset+intLen-len, offset+intLen), sign
719             );
720     }
721
722     /**
723     * Discards all ints whose index is greater than {@code n}.
724     */
725     private void keepLower(int n) {
726         if (intLen >= n) {
727             offset += intLen - n;
728             intLen = n;
729         }
730     }
731
732     /**
733     * Adds the contents of two MutableBigInteger objects. The result
734     * is placed within this MutableBigInteger.
735     * The contents of the addend are not changed.
736     */
737     void add(MutableBigInteger addend) {
738         int x = intLen;
739         int y = addend.intLen;
740         int resultLen = (intLen > addend.intLen ? intLen : addend.intLen);
741         int[] result = (value.length < resultLen ? new int[resultLen] : value);
742
743         int rstart = result.length-1;
744         long sum;
745         long carry = 0;
746
747         // Add common parts of both numbers
748         while(x > 0 && y > 0) {
749             x--; y--;
750             sum = (value[x+offset] & LONG_MASK) +
751                 (addend.value[y+addend.offset] & LONG_MASK) + carry;
752             result[rstart--] = (int)sum;
753             carry = sum >>> 32;
754         }
755
756         // Add remainder of the longer number
757         while(x > 0) {
758             x--;
759             if (carry == 0 && result == value && rstart == (x + offset))
760                 return;
761             sum = (value[x+offset] & LONG_MASK) + carry;

```

```

762         result[rstart--] = (int)sum;
763         carry = sum >>> 32;
764     }
765     while(y > 0) {
766         y--;
767         sum = (addend.value[y+addend.offset] & LONG_MASK) + carry;
768         result[rstart--] = (int)sum;
769         carry = sum >>> 32;
770     }
771
772     if (carry > 0) { // Result must grow in length
773         resultLen++;
774         if (result.length < resultLen) {
775             int temp[] = new int[resultLen];
776             // Result one word longer from carry-out; copy low-order
777             // bits into new result.
778             System.arraycopy(result, 0, temp, 1, result.length);
779             temp[0] = 1;
780             result = temp;
781         } else {
782             result[rstart--] = 1;
783         }
784     }
785
786     value = result;
787     intLen = resultLen;
788     offset = result.length - resultLen;
789 }
790
791 /**
792  * Adds the value of {@code addend} shifted {@code n} ints to the left.
793  * Has the same effect as {@code addend.leftShift(32*ints); add(addend);}
794  * but doesn't change the value of {@code addend}.
795  */
796 void addShifted(MutableBigInteger addend, int n) {
797     if (addend.isZero()) {
798         return;
799     }
800
801     int x = intLen;
802     int y = addend.intLen + n;
803     int resultLen = (intLen > y ? intLen : y);
804     int[] result = (value.length < resultLen ? new int[resultLen] : value);
805
806     int rstart = result.length-1;
807     long sum;
808     long carry = 0;
809
810     // Add common parts of both numbers
811     while (x > 0 && y > 0) {
812         x--; y--;
813         int bval = y+addend.offset < addend.value.length ? addend.value[y+addend.offset] : 0;
814         sum = (value[x+offset] & LONG_MASK) +

```

```

815         (bval & LONG_MASK) + carry;
816         result[rstart--] = (int)sum;
817         carry = sum >>> 32;
818     }
819
820     // Add remainder of the longer number
821     while (x > 0) {
822         x--;
823         if (carry == 0 && result == value && rstart == (x + offset)) {
824             return;
825         }
826         sum = (value[x+offset] & LONG_MASK) + carry;
827         result[rstart--] = (int)sum;
828         carry = sum >>> 32;
829     }
830     while (y > 0) {
831         y--;
832         int bval = y+addend.offset < addend.value.length ? addend.value[y+addend.offset] : 0;
833         sum = (bval & LONG_MASK) + carry;
834         result[rstart--] = (int)sum;
835         carry = sum >>> 32;
836     }
837
838     if (carry > 0) { // Result must grow in length
839         resultLen++;
840         if (result.length < resultLen) {
841             int temp[] = new int[resultLen];
842             // Result one word longer from carry-out; copy low-order
843             // bits into new result.
844             System.arraycopy(result, 0, temp, 1, result.length);
845             temp[0] = 1;
846             result = temp;
847         } else {
848             result[rstart--] = 1;
849         }
850     }
851
852     value = result;
853     intLen = resultLen;
854     offset = result.length - resultLen;
855 }
856
857 /**
858  * Like {@link #addShifted(MutableBigInteger, int)} but {@code this.intLen} must
859  * not be greater than {@code n}. In other words, concatenates {@code this}
860  * and {@code addend}.
861  */
862 void addDisjoint(MutableBigInteger addend, int n) {
863     if (addend.isZero())
864         return;
865
866     int x = intLen;
867     int y = addend.intLen + n;

```

```

868     int resultLen = (intLen > y ? intLen : y);
869     int[] result;
870     if (value.length < resultLen)
871         result = new int[resultLen];
872     else {
873         result = value;
874         Arrays.fill(value, offset+intLen, value.length, 0);
875     }
876
877     int rstart = result.length-1;
878
879     // copy from this if needed
880     System.arraycopy(value, offset, result, rstart+1-x, x);
881     y -= x;
882     rstart -= x;
883
884     int len = Math.min(y, addend.value.length-addend.offset);
885     System.arraycopy(addend.value, addend.offset, result, rstart+1-y, len);
886
887     // zero the gap
888     for (int i=rstart+1-y+len; i < rstart+1; i++)
889         result[i] = 0;
890
891     value = result;
892     intLen = resultLen;
893     offset = result.length - resultLen;
894 }
895
896 /**
897  * Adds the low {@code n} ints of {@code addend}.
898  */
899 void addLower(MutableBigInteger addend, int n) {
900     MutableBigInteger a = new MutableBigInteger(addend);
901     if (a.offset + a.intLen >= n) {
902         a.offset = a.offset + a.intLen - n;
903         a.intLen = n;
904     }
905     a.normalize();
906     add(a);
907 }
908
909 /**
910  * Subtracts the smaller of this and b from the larger and places the
911  * result into this MutableBigInteger.
912  */
913 int subtract(MutableBigInteger b) {
914     MutableBigInteger a = this;
915
916     int[] result = value;
917     int sign = a.compare(b);
918
919     if (sign == 0) {
920         reset();

```

```

921         return 0;
922     }
923     if (sign < 0) {
924         MutableBigInteger tmp = a;
925         a = b;
926         b = tmp;
927     }
928
929     int resultLen = a.intLen;
930     if (result.length < resultLen)
931         result = new int[resultLen];
932
933     long diff = 0;
934     int x = a.intLen;
935     int y = b.intLen;
936     int rstart = result.length - 1;
937
938     // Subtract common parts of both numbers
939     while (y > 0) {
940         x--; y--;
941
942         diff = (a.value[x+a.offset] & LONG_MASK) -
943             (b.value[y+b.offset] & LONG_MASK) - ((int)-(diff>>32));
944         result[rstart--] = (int)diff;
945     }
946     // Subtract remainder of longer number
947     while (x > 0) {
948         x--;
949         diff = (a.value[x+a.offset] & LONG_MASK) - ((int)-(diff>>32));
950         result[rstart--] = (int)diff;
951     }
952
953     value = result;
954     intLen = resultLen;
955     offset = value.length - resultLen;
956     normalize();
957     return sign;
958 }
959
960 /**
961  * Subtracts the smaller of a and b from the larger and places the result
962  * into the larger. Returns 1 if the answer is in a, -1 if in b, 0 if no
963  * operation was performed.
964  */
965 private int difference(MutableBigInteger b) {
966     MutableBigInteger a = this;
967     int sign = a.compare(b);
968     if (sign == 0)
969         return 0;
970     if (sign < 0) {
971         MutableBigInteger tmp = a;
972         a = b;
973         b = tmp;

```



```

974     }
975
976     long diff = 0;
977     int x = a.intLen;
978     int y = b.intLen;
979
980     // Subtract common parts of both numbers
981     while (y > 0) {
982         x--; y--;
983         diff = (a.value[a.offset+ x] & LONG_MASK) -
984             (b.value[b.offset+ y] & LONG_MASK) - ((int)-(diff>>32));
985         a.value[a.offset+x] = (int)diff;
986     }
987     // Subtract remainder of longer number
988     while (x > 0) {
989         x--;
990         diff = (a.value[a.offset+ x] & LONG_MASK) - ((int)-(diff>>32));
991         a.value[a.offset+x] = (int)diff;
992     }
993
994     a.normalize();
995     return sign;
996 }
997
998 /**
999  * Multiply the contents of two MutableBigInteger objects. The result is
1000  * placed into MutableBigInteger z. The contents of y are not changed.
1001  */
1002 void multiply(MutableBigInteger y, MutableBigInteger z) {
1003     int xLen = intLen;
1004     int yLen = y.intLen;
1005     int newLen = xLen + yLen;
1006
1007     // Put z into an appropriate state to receive product
1008     if (z.value.length < newLen)
1009         z.value = new int[newLen];
1010     z.offset = 0;
1011     z.intLen = newLen;
1012
1013     // The first iteration is hoisted out of the loop to avoid extra add
1014     long carry = 0;
1015     for (int j=yLen-1, k=yLen+xLen-1; j >= 0; j--, k--) {
1016         long product = (y.value[j+y.offset] & LONG_MASK) *
1017             (value[xLen-1+offset] & LONG_MASK) + carry;
1018         z.value[k] = (int)product;
1019         carry = product >>> 32;
1020     }
1021     z.value[xLen-1] = (int)carry;
1022
1023     // Perform the multiplication word by word
1024     for (int i = xLen-2; i >= 0; i--) {
1025         carry = 0;
1026         for (int j=yLen-1, k=yLen+i; j >= 0; j--, k--) {

```

```

1027         long product = (y.value[j+y.offset] & LONG_MASK) *
1028             (value[i+offset] & LONG_MASK) +
1029             (z.value[k] & LONG_MASK) + carry;
1030         z.value[k] = (int)product;
1031         carry = product >>> 32;
1032     }
1033     z.value[i] = (int)carry;
1034 }
1035
1036 // Remove leading zeros from product
1037 z.normalize();
1038 }
1039
1040 /**
1041  * Multiply the contents of this MutableBigInteger by the word y. The
1042  * result is placed into z.
1043  */
1044 void mul(int y, MutableBigInteger z) {
1045     if (y == 1) {
1046         z.copyValue(this);
1047         return;
1048     }
1049
1050     if (y == 0) {
1051         z.clear();
1052         return;
1053     }
1054
1055     // Perform the multiplication word by word
1056     long ylong = y & LONG_MASK;
1057     int[] zval = (z.value.length < intLen+1 ? new int[intLen + 1]
1058         : z.value);
1059
1060     long carry = 0;
1061     for (int i = intLen-1; i >= 0; i--) {
1062         long product = ylong * (value[i+offset] & LONG_MASK) + carry;
1063         zval[i+1] = (int)product;
1064         carry = product >>> 32;
1065     }
1066
1067     if (carry == 0) {
1068         z.offset = 1;
1069         z.intLen = intLen;
1070     } else {
1071         z.offset = 0;
1072         z.intLen = intLen + 1;
1073         zval[0] = (int)carry;
1074     }
1075     z.value = zval;
1076 }
1077
1078 /**
1079  * This method is used for division of an n word dividend by a one word
1080  * divisor. The quotient is placed into quotient. The one word divisor is

```

```

1080     * specified by divisor.
1081     *
1082     * @return the remainder of the division is returned.
1083     *
1084     */
1085     int divideOneWord(int divisor, MutableBigInteger quotient) {
1086         long divisorLong = divisor & LONG_MASK;
1087
1088         // Special case of one word dividend
1089         if (intLen == 1) {
1090             long dividendValue = value[offset] & LONG_MASK;
1091             int q = (int) (dividendValue / divisorLong);
1092             int r = (int) (dividendValue - q * divisorLong);
1093             quotient.value[0] = q;
1094             quotient.intLen = (q == 0) ? 0 : 1;
1095             quotient.offset = 0;
1096             return r;
1097         }
1098
1099         if (quotient.value.length < intLen)
1100             quotient.value = new int[intLen];
1101         quotient.offset = 0;
1102         quotient.intLen = intLen;
1103
1104         // Normalize the divisor
1105         int shift = Integer.numberOfLeadingZeros(divisor);
1106
1107         int rem = value[offset];
1108         long remLong = rem & LONG_MASK;
1109         if (remLong < divisorLong) {
1110             quotient.value[0] = 0;
1111         } else {
1112             quotient.value[0] = (int)(remLong / divisorLong);
1113             rem = (int) (remLong - (quotient.value[0] * divisorLong));
1114             remLong = rem & LONG_MASK;
1115         }
1116         int xlen = intLen;
1117         while (--xlen > 0) {
1118             long dividendEstimate = (remLong << 32) |
1119                                     (value[offset + intLen - xlen] & LONG_MASK);
1120             int q;
1121             if (dividendEstimate >= 0) {
1122                 q = (int) (dividendEstimate / divisorLong);
1123                 rem = (int) (dividendEstimate - q * divisorLong);
1124             } else {
1125                 long tmp = divWord(dividendEstimate, divisor);
1126                 q = (int) (tmp & LONG_MASK);
1127                 rem = (int) (tmp >>> 32);
1128             }
1129             quotient.value[intLen - xlen] = q;
1130             remLong = rem & LONG_MASK;
1131         }
1132     }

```

```

1133     quotient.normalize();
1134     // Unnormalize
1135     if (shift > 0)
1136         return rem % divisor;
1137     else
1138         return rem;
1139 }
1140
1141 /**
1142  * Calculates the quotient of this div b and places the quotient in the
1143  * provided MutableBigInteger objects and the remainder object is returned.
1144  *
1145  */
1146 MutableBigInteger divide(MutableBigInteger b, MutableBigInteger quotient) {
1147     return divide(b, quotient, true);
1148 }
1149
1150 MutableBigInteger divide(MutableBigInteger b, MutableBigInteger quotient, boolean needRemainder
1151     ) {
1152     if (b.intLen < BigInteger.BURNIKEL_ZIEGLER_THRESHOLD ||
1153         intLen - b.intLen < BigInteger.BURNIKEL_ZIEGLER_OFFSET) {
1154         return divideKnuth(b, quotient, needRemainder);
1155     } else {
1156         return divideAndRemainderBurnikelZiegler(b, quotient);
1157     }
1158 }
1159
1160 /**
1161  * @see #divideKnuth(MutableBigInteger, MutableBigInteger, boolean)
1162  */
1163 MutableBigInteger divideKnuth(MutableBigInteger b, MutableBigInteger quotient) {
1164     return divideKnuth(b, quotient, true);
1165 }
1166
1167 /**
1168  * Calculates the quotient of this div b and places the quotient in the
1169  * provided MutableBigInteger objects and the remainder object is returned.
1170  *
1171  * Uses Algorithm D in Knuth section 4.3.1.
1172  * Many optimizations to that algorithm have been adapted from the Colin
1173  * Plumb C library.
1174  * It special cases one word divisors for speed. The content of b is not
1175  * changed.
1176  *
1177  */
1178 MutableBigInteger divideKnuth(MutableBigInteger b, MutableBigInteger quotient, boolean
1179     needRemainder) {
1180     if (b.intLen == 0)
1181         throw new ArithmeticException("BigInteger divide by zero");
1182
1183     // Dividend is zero
1184     if (intLen == 0) {
1185         quotient.intLen = quotient.offset = 0;

```

```

1184         return needRemainder ? new MutableBigInteger() : null;
1185     }
1186
1187     int cmp = compare(b);
1188     // Dividend less than divisor
1189     if (cmp < 0) {
1190         quotient.intLen = quotient.offset = 0;
1191         return needRemainder ? new MutableBigInteger(this) : null;
1192     }
1193     // Dividend equal to divisor
1194     if (cmp == 0) {
1195         quotient.value[0] = quotient.intLen = 1;
1196         quotient.offset = 0;
1197         return needRemainder ? new MutableBigInteger() : null;
1198     }
1199
1200     quotient.clear();
1201     // Special case one word divisor
1202     if (b.intLen == 1) {
1203         int r = divideOneWord(b.value[b.offset], quotient);
1204         if (needRemainder) {
1205             if (r == 0)
1206                 return new MutableBigInteger();
1207             return new MutableBigInteger(r);
1208         } else {
1209             return null;
1210         }
1211     }
1212
1213     // Cancel common powers of two if we're above the KNUTH_POW2_* thresholds
1214     if (intLen >= KNUTH_POW2_THRESH_LEN) {
1215         int trailingZeroBits = Math.min(getLowestSetBit(), b.getLowestSetBit());
1216         if (trailingZeroBits >= KNUTH_POW2_THRESH_ZEROS*32) {
1217             MutableBigInteger a = new MutableBigInteger(this);
1218             b = new MutableBigInteger(b);
1219             a.rightShift(trailingZeroBits);
1220             b.rightShift(trailingZeroBits);
1221             MutableBigInteger r = a.divideKnuth(b, quotient);
1222             r.leftShift(trailingZeroBits);
1223             return r;
1224         }
1225     }
1226
1227     return divideMagnitude(b, quotient, needRemainder);
1228 }
1229
1230 /**
1231  * Computes {@code this/b} and {@code this%b} using the
1232  * Burnikel-Ziegler algorithm.
1233  * This method implements algorithm 3 from pg. 9 of the Burnikel-Ziegler paper.
1234  * The parameter beta was chosen to be  $2^{32}$  so almost all shifts are
1235  * multiples of 32 bits.
1236  * {@code this} and {@code b} must be nonnegative.

```

```

1237     * @param b the divisor
1238     * @param quotient output parameter for {@code this/b}
1239     * @return the remainder
1240     */
1241     MutableBigInteger divideAndRemainderBurnikelZiegler(MutableBigInteger b, MutableBigInteger
        quotient) {
1242         int r = intLen;
1243         int s = b.intLen;
1244
1245         // Clear the quotient
1246         quotient.offset = quotient.intLen = 0;
1247
1248         if (r < s) {
1249             return this;
1250         } else {
1251             // Unlike Knuth division, we don't check for common powers of two here because
1252             // BZ already runs faster if both numbers contain powers of two and cancelling them has
1253             // no
1254             // additional benefit.
1255
1256             // step 1: let m = min{2^k | (2^k)*BURNIKEL_ZIEGLER_THRESHOLD > s}
1257             int m = 1 << (32-Integer.numberOfLeadingZeros(s/BigInteger.BURNIKEL_ZIEGLER_THRESHOLD))
                ;
1258
1259             int j = (s+m-1) / m;          // step 2a: j = ceil(s/m)
1260             int n = j * m;                // step 2b: block length in 32-bit units
1261             long n32 = 32L * n;           // block length in bits
1262             int sigma = (int) Math.max(0, n32 - b.bitLength()); // step 3: sigma = max{T | (2^T)*
                B < beta^n}
1263             MutableBigInteger bShifted = new MutableBigInteger(b);
1264             bShifted.safeLeftShift(sigma); // step 4a: shift b so its length is a multiple of n
1265             MutableBigInteger aShifted = new MutableBigInteger(this);
1266             aShifted.safeLeftShift(sigma); // step 4b: shift a by the same amount
1267
1268             // step 5: t is the number of blocks needed to accommodate a plus one additional bit
1269             int t = (int) ((aShifted.bitLength()+n32) / n32);
1270             if (t < 2) {
1271                 t = 2;
1272             }
1273
1274             // step 6: conceptually split a into blocks a[t-1], ..., a[0]
1275             MutableBigInteger a1 = aShifted.getBlock(t-1, t, n); // the most significant block of
                a
1276
1277             // step 7: z[t-2] = [a[t-1], a[t-2]]
1278             MutableBigInteger z = aShifted.getBlock(t-2, t, n); // the second to most
                significant block
1279             z.addDisjoint(a1, n); // z[t-2]
1280
1281             // do schoolbook division on blocks, dividing 2-block numbers by 1-block numbers
1282             MutableBigInteger qi = new MutableBigInteger();
1283             MutableBigInteger ri;
1284             for (int i=t-2; i > 0; i--) {

```

```

1284         // step 8a: compute (qi,ri) such that z=b*qi+ri
1285         ri = z.divide2n1n(bShifted, qi);
1286
1287         // step 8b: z = [ri, a[i-1]]
1288         z = aShifted.getBlock(i-1, t, n); // a[i-1]
1289         z.addDisjoint(ri, n);
1290         quotient.addShifted(qi, i*n); // update q (part of step 9)
1291     }
1292     // final iteration of step 8: do the loop one more time for i=0 but leave z unchanged
1293     ri = z.divide2n1n(bShifted, qi);
1294     quotient.add(qi);
1295
1296     ri.rightShift(sigma); // step 9: a and b were shifted, so shift back
1297     return ri;
1298 }
1299 }
1300
1301 /**
1302  * This method implements algorithm 1 from pg. 4 of the Burnikel-Ziegler paper.
1303  * It divides a 2n-digit number by a n-digit number.<br/>
1304  * The parameter beta is  $2^{32}$  so all shifts are multiples of 32 bits.
1305  * <br/>
1306  * {@code this} must be a nonnegative number such that {@code this.bitLength() <= 2*b.bitLength() }
1307  * @param b a positive number such that {@code b.bitLength()} is even
1308  * @param quotient output parameter for {@code this/b}
1309  * @return {@code this/b}
1310  */
1311 private MutableBigInteger divide2n1n(MutableBigInteger b, MutableBigInteger quotient) {
1312     int n = b.intLen;
1313
1314     // step 1: base case
1315     if (n%2 != 0 || n < BigInteger.BURNIKEL_ZIEGLER_THRESHOLD) {
1316         return divideKnuth(b, quotient);
1317     }
1318
1319     // step 2: view this as [a1,a2,a3,a4] where each ai is n/2 ints or less
1320     MutableBigInteger aUpper = new MutableBigInteger(this);
1321     aUpper.safeRightShift(32*(n/2)); // aUpper = [a1,a2,a3]
1322     keepLower(n/2); // this = a4
1323
1324     // step 3: q1=aUpper/b, r1=aUpper%b
1325     MutableBigInteger q1 = new MutableBigInteger();
1326     MutableBigInteger r1 = aUpper.divide3n2n(b, q1);
1327
1328     // step 4: quotient=[r1,this]/b, r2=[r1,this]%b
1329     addDisjoint(r1, n/2); // this = [r1,this]
1330     MutableBigInteger r2 = divide3n2n(b, quotient);
1331
1332     // step 5: let quotient=[q1,quotient] and return r2
1333     quotient.addDisjoint(q1, n/2);
1334     return r2;
1335 }

```

```

1336
1337 /**
1338  * This method implements algorithm 2 from pg. 5 of the Burnikel-Ziegler paper.
1339  * It divides a 3n-digit number by a 2n-digit number.<br/>
1340  * The parameter beta is  $2^{32}$  so all shifts are multiples of 32 bits.<br/>
1341  * <br/>
1342  * {@code this} must be a nonnegative number such that {@code 2*this.bitLength() <= 3*b.
        bitLength()}
1343  * @param quotient output parameter for {@code this/b}
1344  * @return {@code this/b}
1345  */
1346 private MutableBigInteger divide3n2n(MutableBigInteger b, MutableBigInteger quotient) {
1347     int n = b.intLen / 2;    // half the length of b in ints
1348
1349     // step 1: view this as [a1,a2,a3] where each ai is n ints or less; let a12=[a1,a2]
1350     MutableBigInteger a12 = new MutableBigInteger(this);
1351     a12.safeRightShift(32*n);
1352
1353     // step 2: view b as [b1,b2] where each bi is n ints or less
1354     MutableBigInteger b1 = new MutableBigInteger(b);
1355     b1.safeRightShift(n * 32);
1356     BigInteger b2 = b.getLower(n);
1357
1358     MutableBigInteger r;
1359     MutableBigInteger d;
1360     if (compareShifted(b, n) < 0) {
1361         // step 3a: if a1<b1, let quotient=a12/b1 and r=a12%b1
1362         r = a12.divide2n1n(b1, quotient);
1363
1364         // step 4: d=quotient*b2
1365         d = new MutableBigInteger(quotient.toBigInteger().multiply(b2));
1366     } else {
1367         // step 3b: if a1>=b1, let quotient=beta^n-1 and r=a12-b1*2^n+b1
1368         quotient.ones(n);
1369         a12.add(b1);
1370         b1.leftShift(32*n);
1371         a12.subtract(b1);
1372         r = a12;
1373
1374         // step 4: d=quotient*b2=(b2 << 32*n) - b2
1375         d = new MutableBigInteger(b2);
1376         d.leftShift(32 * n);
1377         d.subtract(new MutableBigInteger(b2));
1378     }
1379
1380     // step 5: r = r*beta^n + a3 - d (paper says a4)
1381     // However, don't subtract d until after the while loop so r doesn't become negative
1382     r.leftShift(32 * n);
1383     r.addLower(this, n);
1384
1385     // step 6: add b until r>=d
1386     while (r.compare(d) < 0) {
1387         r.add(b);

```



```

1388     quotient.subtract(MutableBigInteger.ONE);
1389 }
1390 r.subtract(d);
1391
1392     return r;
1393 }
1394
1395 /**
1396  * Returns a {@code MutableBigInteger} containing {@code blockLength} ints from
1397  * {@code this} number, starting at {@code index*blockLength}.<br/>
1398  * Used by Burnikel-Ziegler division.
1399  * @param index the block index
1400  * @param numBlocks the total number of blocks in {@code this} number
1401  * @param blockLength length of one block in units of 32 bits
1402  * @return
1403  */
1404 private MutableBigInteger getBlock(int index, int numBlocks, int blockLength) {
1405     int blockStart = index * blockLength;
1406     if (blockStart >= intLen) {
1407         return new MutableBigInteger();
1408     }
1409
1410     int blockEnd;
1411     if (index == numBlocks-1) {
1412         blockEnd = intLen;
1413     } else {
1414         blockEnd = (index+1) * blockLength;
1415     }
1416     if (blockEnd > intLen) {
1417         return new MutableBigInteger();
1418     }
1419
1420     int[] newVal = Arrays.copyOfRange(value, offset+intLen-blockEnd, offset+intLen-blockStart);
1421     return new MutableBigInteger(newVal);
1422 }
1423
1424 /** @see BigInteger#bitLength() */
1425 long bitLength() {
1426     if (intLen == 0)
1427         return 0;
1428     return intLen*32L - Integer.numberOfLeadingZeros(value[offset]);
1429 }
1430
1431 /**
1432  * Internally used to calculate the quotient of this div v and places the
1433  * quotient in the provided MutableBigInteger object and the remainder is
1434  * returned.
1435  *
1436  * @return the remainder of the division will be returned.
1437  */
1438 long divide(long v, MutableBigInteger quotient) {
1439     if (v == 0)
1440         throw new ArithmeticException("BigInteger divide by zero");

```

```

1441
1442     // Dividend is zero
1443     if (intLen == 0) {
1444         quotient.intLen = quotient.offset = 0;
1445         return 0;
1446     }
1447     if (v < 0)
1448         v = -v;
1449
1450     int d = (int)(v >>> 32);
1451     quotient.clear();
1452     // Special case on word divisor
1453     if (d == 0)
1454         return divideOneWord((int)v, quotient) & LONG_MASK;
1455     else {
1456         return divideLongMagnitude(v, quotient).toLong();
1457     }
1458 }
1459
1460 private static void copyAndShift(int[] src, int srcFrom, int srcLen, int[] dst, int dstFrom,
1461     int shift) {
1462     int n2 = 32 - shift;
1463     int c=src[srcFrom];
1464     for (int i=0; i < srcLen-1; i++) {
1465         int b = c;
1466         c = src[++srcFrom];
1467         dst[dstFrom+i] = (b << shift) | (c >>> n2);
1468     }
1469     dst[dstFrom+srcLen-1] = c << shift;
1470 }
1471
1472 /**
1473  * Divide this MutableBigInteger by the divisor.
1474  * The quotient will be placed into the provided quotient object &
1475  * the remainder object is returned.
1476  */
1477 private MutableBigInteger divideMagnitude(MutableBigInteger div,
1478     MutableBigInteger quotient,
1479     boolean needRemainder ) {
1480     // assert div.intLen > 1
1481     // D1 normalize the divisor
1482     int shift = Integer.numberOfLeadingZeros(div.value[div.offset]);
1483     // Copy divisor value to protect divisor
1484     final int dlen = div.intLen;
1485     int[] divisor;
1486     MutableBigInteger rem; // Remainder starts as dividend with space for a leading zero
1487     if (shift > 0) {
1488         divisor = new int[dlen];
1489         copyAndShift(div.value,div.offset,dlen,divisor,0,shift);
1490         if (Integer.numberOfLeadingZeros(value[offset]) >= shift) {
1491             int[] remarr = new int[intLen + 1];
1492             rem = new MutableBigInteger(remarr);
1493             rem.intLen = intLen;

```

```

1493         rem.offset = 1;
1494         copyAndShift(value,offset,intLen,remarr,1,shift);
1495     } else {
1496         int[] remarr = new int[intLen + 2];
1497         rem = new MutableBigInteger(remarr);
1498         rem.intLen = intLen+1;
1499         rem.offset = 1;
1500         int rFrom = offset;
1501         int c=0;
1502         int n2 = 32 - shift;
1503         for (int i=1; i < intLen+1; i++,rFrom++) {
1504             int b = c;
1505             c = value[rFrom];
1506             remarr[i] = (b << shift) | (c >>> n2);
1507         }
1508         remarr[intLen+1] = c << shift;
1509     }
1510 } else {
1511     divisor = Arrays.copyOfRange(div.value, div.offset, div.offset + div.intLen);
1512     rem = new MutableBigInteger(new int[intLen + 1]);
1513     System.arraycopy(value, offset, rem.value, 1, intLen);
1514     rem.intLen = intLen;
1515     rem.offset = 1;
1516 }
1517
1518 int nlen = rem.intLen;
1519
1520 // Set the quotient size
1521 final int limit = nlen - dlen + 1;
1522 if (quotient.value.length < limit) {
1523     quotient.value = new int[limit];
1524     quotient.offset = 0;
1525 }
1526 quotient.intLen = limit;
1527 int[] q = quotient.value;
1528
1529
1530 // Must insert leading 0 in rem if its length did not change
1531 if (rem.intLen == nlen) {
1532     rem.offset = 0;
1533     rem.value[0] = 0;
1534     rem.intLen++;
1535 }
1536
1537 int dh = divisor[0];
1538 long dhLong = dh & LONG_MASK;
1539 int dl = divisor[1];
1540
1541 // D2 Initialize j
1542 for (int j=0; j < limit-1; j++) {
1543     // D3 Calculate qhat
1544     // estimate qhat
1545     int qhat = 0;

```

```

1546     int qrem = 0;
1547     boolean skipCorrection = false;
1548     int nh = rem.value[j+rem.offset];
1549     int nh2 = nh + 0x80000000;
1550     int nm = rem.value[j+1+rem.offset];
1551
1552     if (nh == dh) {
1553         qhat = ~0;
1554         qrem = nh + nm;
1555         skipCorrection = qrem + 0x80000000 < nh2;
1556     } else {
1557         long nChunk = (((long)nh) << 32) | (nm & LONG_MASK);
1558         if (nChunk >= 0) {
1559             qhat = (int) (nChunk / dhLong);
1560             qrem = (int) (nChunk - (qhat * dhLong));
1561         } else {
1562             long tmp = divWord(nChunk, dh);
1563             qhat = (int) (tmp & LONG_MASK);
1564             qrem = (int) (tmp >>> 32);
1565         }
1566     }
1567
1568     if (qhat == 0)
1569         continue;
1570
1571     if (!skipCorrection) { // Correct qhat
1572         long nl = rem.value[j+2+rem.offset] & LONG_MASK;
1573         long rs = ((qrem & LONG_MASK) << 32) | nl;
1574         long estProduct = (dl & LONG_MASK) * (qhat & LONG_MASK);
1575
1576         if (unsignedLongCompare(estProduct, rs)) {
1577             qhat--;
1578             qrem = (int)((qrem & LONG_MASK) + dhLong);
1579             if ((qrem & LONG_MASK) >= dhLong) {
1580                 estProduct -= (dl & LONG_MASK);
1581                 rs = ((qrem & LONG_MASK) << 32) | nl;
1582                 if (unsignedLongCompare(estProduct, rs))
1583                     qhat--;
1584             }
1585         }
1586     }
1587
1588     // D4 Multiply and subtract
1589     rem.value[j+rem.offset] = 0;
1590     int borrow = mulsub(rem.value, divisor, qhat, dlen, j+rem.offset);
1591
1592     // D5 Test remainder
1593     if (borrow + 0x80000000 > nh2) {
1594         // D6 Add back
1595         divadd(divisor, rem.value, j+1+rem.offset);
1596         qhat--;
1597     }
1598

```

```

1599         // Store the quotient digit
1600         q[j] = qhat;
1601     } // D7 loop on j
1602     // D3 Calculate qhat
1603     // estimate qhat
1604     int qhat = 0;
1605     int qrem = 0;
1606     boolean skipCorrection = false;
1607     int nh = rem.value[limit - 1 + rem.offset];
1608     int nh2 = nh + 0x80000000;
1609     int nm = rem.value[limit + rem.offset];
1610
1611     if (nh == dh) {
1612         qhat = ~0;
1613         qrem = nh + nm;
1614         skipCorrection = qrem + 0x80000000 < nh2;
1615     } else {
1616         long nChunk = (((long) nh) << 32) | (nm & LONG_MASK);
1617         if (nChunk >= 0) {
1618             qhat = (int) (nChunk / dhLong);
1619             qrem = (int) (nChunk - (qhat * dhLong));
1620         } else {
1621             long tmp = divWord(nChunk, dh);
1622             qhat = (int) (tmp & LONG_MASK);
1623             qrem = (int) (tmp >>> 32);
1624         }
1625     }
1626     if (qhat != 0) {
1627         if (!skipCorrection) { // Correct qhat
1628             long nl = rem.value[limit + 1 + rem.offset] & LONG_MASK;
1629             long rs = ((qrem & LONG_MASK) << 32) | nl;
1630             long estProduct = (dl & LONG_MASK) * (qhat & LONG_MASK);
1631
1632             if (unsignedLongCompare(estProduct, rs)) {
1633                 qhat--;
1634                 qrem = (int) ((qrem & LONG_MASK) + dhLong);
1635                 if ((qrem & LONG_MASK) >= dhLong) {
1636                     estProduct -= (dl & LONG_MASK);
1637                     rs = ((qrem & LONG_MASK) << 32) | nl;
1638                     if (unsignedLongCompare(estProduct, rs))
1639                         qhat--;
1640                 }
1641             }
1642         }
1643
1644         // D4 Multiply and subtract
1645         int borrow;
1646         rem.value[limit - 1 + rem.offset] = 0;
1647         if (needRemainder)
1648             borrow = mulsub(rem.value, divisor, qhat, dlen, limit - 1 + rem.offset);
1649         else
1650             borrow = mulsubBorrow(rem.value, divisor, qhat, dlen, limit - 1 + rem.offset);

```

```

1652
1653     // D5 Test remainder
1654     if (borrow + 0x80000000 > nh2) {
1655         // D6 Add back
1656         if (needRemainder)
1657             divadd(divisor, rem.value, limit - 1 + 1 + rem.offset);
1658         qhat--;
1659     }
1660
1661     // Store the quotient digit
1662     q[(limit - 1)] = qhat;
1663 }
1664
1665
1666     if (needRemainder) {
1667         // D8 Unnormalize
1668         if (shift > 0)
1669             rem.rightShift(shift);
1670         rem.normalize();
1671     }
1672     quotient.normalize();
1673     return needRemainder ? rem : null;
1674 }
1675
1676 /**
1677  * Divide this MutableBigInteger by the divisor represented by positive long
1678  * value. The quotient will be placed into the provided quotient object &
1679  * the remainder object is returned.
1680  */
1681 private MutableBigInteger divideLongMagnitude(long ldivisor, MutableBigInteger quotient) {
1682     // Remainder starts as dividend with space for a leading zero
1683     MutableBigInteger rem = new MutableBigInteger(new int[intLen + 1]);
1684     System.arraycopy(value, offset, rem.value, 1, intLen);
1685     rem.intLen = intLen;
1686     rem.offset = 1;
1687
1688     int nlen = rem.intLen;
1689
1690     int limit = nlen - 2 + 1;
1691     if (quotient.value.length < limit) {
1692         quotient.value = new int[limit];
1693         quotient.offset = 0;
1694     }
1695     quotient.intLen = limit;
1696     int[] q = quotient.value;
1697
1698     // D1 normalize the divisor
1699     int shift = Long.numberOfLeadingZeros(ldivisor);
1700     if (shift > 0) {
1701         ldivisor<=<shift;
1702         rem.leftShift(shift);
1703     }
1704

```

```

1705     // Must insert leading 0 in rem if its length did not change
1706     if (rem.intLen == nlen) {
1707         rem.offset = 0;
1708         rem.value[0] = 0;
1709         rem.intLen++;
1710     }
1711
1712     int dh = (int)(ldivisor >>> 32);
1713     long dhLong = dh & LONG_MASK;
1714     int dl = (int)(ldivisor & LONG_MASK);
1715
1716     // D2 Initialize j
1717     for (int j = 0; j < limit; j++) {
1718         // D3 Calculate qhat
1719         // estimate qhat
1720         int qhat = 0;
1721         int qrem = 0;
1722         boolean skipCorrection = false;
1723         int nh = rem.value[j + rem.offset];
1724         int nh2 = nh + 0x80000000;
1725         int nm = rem.value[j + 1 + rem.offset];
1726
1727         if (nh == dh) {
1728             qhat = ~0;
1729             qrem = nh + nm;
1730             skipCorrection = qrem + 0x80000000 < nh2;
1731         } else {
1732             long nChunk = (((long) nh) << 32) | (nm & LONG_MASK);
1733             if (nChunk >= 0) {
1734                 qhat = (int) (nChunk / dhLong);
1735                 qrem = (int) (nChunk - (qhat * dhLong));
1736             } else {
1737                 long tmp = divWord(nChunk, dh);
1738                 qhat = (int) (tmp & LONG_MASK);
1739                 qrem = (int) (tmp >>> 32);
1740             }
1741         }
1742
1743         if (qhat == 0)
1744             continue;
1745
1746         if (!skipCorrection) { // Correct qhat
1747             long nl = rem.value[j + 2 + rem.offset] & LONG_MASK;
1748             long rs = ((qrem & LONG_MASK) << 32) | nl;
1749             long estProduct = (dl & LONG_MASK) * (qhat & LONG_MASK);
1750
1751             if (unsignedLongCompare(estProduct, rs)) {
1752                 qhat--;
1753                 qrem = (int) ((qrem & LONG_MASK) + dhLong);
1754                 if ((qrem & LONG_MASK) >= dhLong) {
1755                     estProduct -= (dl & LONG_MASK);
1756                     rs = ((qrem & LONG_MASK) << 32) | nl;
1757                     if (unsignedLongCompare(estProduct, rs))

```

```

1758         qhat--;
1759     }
1760 }
1761 }
1762
1763 // D4 Multiply and subtract
1764 rem.value[j + rem.offset] = 0;
1765 int borrow = mulsubLong(rem.value, dh, dl, qhat, j + rem.offset);
1766
1767 // D5 Test remainder
1768 if (borrow + 0x80000000 > nh2) {
1769     // D6 Add back
1770     divaddLong(dh, dl, rem.value, j + 1 + rem.offset);
1771     qhat--;
1772 }
1773
1774 // Store the quotient digit
1775 q[j] = qhat;
1776 } // D7 loop on j
1777
1778 // D8 Unnormalize
1779 if (shift > 0)
1780     rem.rightShift(shift);
1781
1782 quotient.normalize();
1783 rem.normalize();
1784 return rem;
1785 }
1786
1787 /**
1788  * A primitive used for division by long.
1789  * Specialized version of the method divadd.
1790  * dh is a high part of the divisor, dl is a low part
1791  */
1792 private int divaddLong(int dh, int dl, int[] result, int offset) {
1793     long carry = 0;
1794
1795     long sum = (dl & LONG_MASK) + (result[1+offset] & LONG_MASK);
1796     result[1+offset] = (int)sum;
1797
1798     sum = (dh & LONG_MASK) + (result[offset] & LONG_MASK) + carry;
1799     result[offset] = (int)sum;
1800     carry = sum >>> 32;
1801     return (int)carry;
1802 }
1803
1804 /**
1805  * This method is used for division by long.
1806  * Specialized version of the method sulsub.
1807  * dh is a high part of the divisor, dl is a low part
1808  */
1809 private int mulsubLong(int[] q, int dh, int dl, int x, int offset) {
1810     long xLong = x & LONG_MASK;

```



```

1811     offset += 2;
1812     long product = (dl & LONG_MASK) * xLong;
1813     long difference = q[offset] - product;
1814     q[offset--] = (int)difference;
1815     long carry = (product >>> 32)
1816         + (((difference & LONG_MASK) >
1817             (((~(int)product) & LONG_MASK))) ? 1:0);
1818     product = (dh & LONG_MASK) * xLong + carry;
1819     difference = q[offset] - product;
1820     q[offset--] = (int)difference;
1821     carry = (product >>> 32)
1822         + (((difference & LONG_MASK) >
1823             (((~(int)product) & LONG_MASK))) ? 1:0);
1824     return (int)carry;
1825 }
1826
1827 /**
1828  * Compare two longs as if they were unsigned.
1829  * Returns true iff one is bigger than two.
1830  */
1831 private boolean unsignedLongCompare(long one, long two) {
1832     return (one+Long.MIN_VALUE) > (two+Long.MIN_VALUE);
1833 }
1834
1835 /**
1836  * This method divides a long quantity by an int to estimate
1837  * qhat for two multi precision numbers. It is used when
1838  * the signed value of n is less than zero.
1839  * Returns long value where high 32 bits contain remainder value and
1840  * low 32 bits contain quotient value.
1841  */
1842 static long divWord(long n, int d) {
1843     long dLong = d & LONG_MASK;
1844     long r;
1845     long q;
1846     if (dLong == 1) {
1847         q = (int)n;
1848         r = 0;
1849         return (r << 32) | (q & LONG_MASK);
1850     }
1851
1852     // Approximate the quotient and remainder
1853     q = (n >>> 1) / (dLong >>> 1);
1854     r = n - q*dLong;
1855
1856     // Correct the approximation
1857     while (r < 0) {
1858         r += dLong;
1859         q--;
1860     }
1861     while (r >= dLong) {
1862         r -= dLong;
1863         q++;

```

```

1864     }
1865     // n - q*dLong == r && 0 <= r < dLong, hence we're done.
1866     return (r << 32) | (q & LONG_MASK);
1867 }
1868
1869 /**
1870  * Calculate GCD of this and b. This and b are changed by the computation.
1871  */
1872 MutableBigInteger hybridGCD(MutableBigInteger b) {
1873     // Use Euclid's algorithm until the numbers are approximately the
1874     // same length, then use the binary GCD algorithm to find the GCD.
1875     MutableBigInteger a = this;
1876     MutableBigInteger q = new MutableBigInteger();
1877
1878     while (b.intLen != 0) {
1879         if (Math.abs(a.intLen - b.intLen) < 2)
1880             return a.binaryGCD(b);
1881
1882         MutableBigInteger r = a.divide(b, q);
1883         a = b;
1884         b = r;
1885     }
1886     return a;
1887 }
1888
1889 /**
1890  * Calculate GCD of this and v.
1891  * Assumes that this and v are not zero.
1892  */
1893 private MutableBigInteger binaryGCD(MutableBigInteger v) {
1894     // Algorithm B from Knuth section 4.5.2
1895     MutableBigInteger u = this;
1896     MutableBigInteger r = new MutableBigInteger();
1897
1898     // step B1
1899     int s1 = u.getLowestSetBit();
1900     int s2 = v.getLowestSetBit();
1901     int k = (s1 < s2) ? s1 : s2;
1902     if (k != 0) {
1903         u.rightShift(k);
1904         v.rightShift(k);
1905     }
1906
1907     // step B2
1908     boolean uOdd = (k == s1);
1909     MutableBigInteger t = uOdd ? v : u;
1910     int tsign = uOdd ? -1 : 1;
1911
1912     int lb;
1913     while ((lb = t.getLowestSetBit()) >= 0) {
1914         // steps B3 and B4
1915         t.rightShift(lb);
1916         // step B5

```

```

1917         if (tsign > 0)
1918             u = t;
1919         else
1920             v = t;
1921
1922         // Special case one word numbers
1923         if (u.intLen < 2 && v.intLen < 2) {
1924             int x = u.value[u.offset];
1925             int y = v.value[v.offset];
1926             x = binaryGcd(x, y);
1927             r.value[0] = x;
1928             r.intLen = 1;
1929             r.offset = 0;
1930             if (k > 0)
1931                 r.leftShift(k);
1932             return r;
1933         }
1934
1935         // step B6
1936         if ((tsign = u.difference(v)) == 0)
1937             break;
1938         t = (tsign >= 0) ? u : v;
1939     }
1940
1941     if (k > 0)
1942         u.leftShift(k);
1943     return u;
1944 }
1945
1946 /**
1947  * Calculate GCD of a and b interpreted as unsigned integers.
1948  */
1949 static int binaryGcd(int a, int b) {
1950     if (b == 0)
1951         return a;
1952     if (a == 0)
1953         return b;
1954
1955     // Right shift a & b till their last bits equal to 1.
1956     int aZeros = Integer.numberOfTrailingZeros(a);
1957     int bZeros = Integer.numberOfTrailingZeros(b);
1958     a >>= aZeros;
1959     b >>= bZeros;
1960
1961     int t = (aZeros < bZeros ? aZeros : bZeros);
1962
1963     while (a != b) {
1964         if ((a+0x80000000) > (b+0x80000000)) { // a > b as unsigned
1965             a -= b;
1966             a >>= Integer.numberOfTrailingZeros(a);
1967         } else {
1968             b -= a;
1969             b >>= Integer.numberOfTrailingZeros(b);

```

```

1970     }
1971 }
1972     return a<<t;
1973 }
1974
1975 /**
1976  * Returns the modInverse of this mod p. This and p are not affected by
1977  * the operation.
1978  */
1979 MutableBigInteger mutableModInverse(MutableBigInteger p) {
1980     // Modulus is odd, use Schroeppe's algorithm
1981     if (p.isOdd())
1982         return modInverse(p);
1983
1984     // Base and modulus are even, throw exception
1985     if (isEven())
1986         throw new ArithmeticException("BigInteger not invertible.");
1987
1988     // Get even part of modulus expressed as a power of 2
1989     int powersOf2 = p.getLowestSetBit();
1990
1991     // Construct odd part of modulus
1992     MutableBigInteger oddMod = new MutableBigInteger(p);
1993     oddMod.rightShift(powersOf2);
1994
1995     if (oddMod.isOne())
1996         return modInverseMP2(powersOf2);
1997
1998     // Calculate 1/a mod oddMod
1999     MutableBigInteger oddPart = modInverse(oddMod);
2000
2001     // Calculate 1/a mod evenMod
2002     MutableBigInteger evenPart = modInverseMP2(powersOf2);
2003
2004     // Combine the results using Chinese Remainder Theorem
2005     MutableBigInteger y1 = modInverseBP2(oddMod, powersOf2);
2006     MutableBigInteger y2 = oddMod.modInverseMP2(powersOf2);
2007
2008     MutableBigInteger temp1 = new MutableBigInteger();
2009     MutableBigInteger temp2 = new MutableBigInteger();
2010     MutableBigInteger result = new MutableBigInteger();
2011
2012     oddPart.leftShift(powersOf2);
2013     oddPart.multiply(y1, result);
2014
2015     evenPart.multiply(oddMod, temp1);
2016     temp1.multiply(y2, temp2);
2017
2018     result.add(temp2);
2019     return result.divide(p, temp1);
2020 }
2021
2022 /**

```

```

2023     * Calculate the multiplicative inverse of this mod 2k.
2024     */
2025     MutableBigInteger modInverseMP2(int k) {
2026         if (isEven())
2027             throw new ArithmeticException("Non-invertible. (GCD != 1)");
2028
2029         if (k > 64)
2030             return euclidModInverse(k);
2031
2032         int t = inverseMod32(value[offset+intLen-1]);
2033
2034         if (k < 33) {
2035             t = (k == 32 ? t : t & ((1 << k) - 1));
2036             return new MutableBigInteger(t);
2037         }
2038
2039         long pLong = (value[offset+intLen-1] & LONG_MASK);
2040         if (intLen > 1)
2041             pLong |= ((long)value[offset+intLen-2] << 32);
2042         long tLong = t & LONG_MASK;
2043         tLong = tLong * (2 - pLong * tLong); // 1 more Newton iter step
2044         tLong = (k == 64 ? tLong : tLong & ((1L << k) - 1));
2045
2046         MutableBigInteger result = new MutableBigInteger(new int[2]);
2047         result.value[0] = (int)(tLong >>> 32);
2048         result.value[1] = (int)tLong;
2049         result.intLen = 2;
2050         result.normalize();
2051         return result;
2052     }
2053
2054     /**
2055     * Returns the multiplicative inverse of val mod 232. Assumes val is odd.
2056     */
2057     static int inverseMod32(int val) {
2058         // Newton's iteration!
2059         int t = val;
2060         t *= 2 - val*t;
2061         t *= 2 - val*t;
2062         t *= 2 - val*t;
2063         t *= 2 - val*t;
2064         return t;
2065     }
2066
2067     /**
2068     * Returns the multiplicative inverse of val mod 264. Assumes val is odd.
2069     */
2070     static long inverseMod64(long val) {
2071         // Newton's iteration!
2072         long t = val;
2073         t *= 2 - val*t;
2074         t *= 2 - val*t;
2075         t *= 2 - val*t;

```

```

2076         t *= 2 - val*t;
2077         t *= 2 - val*t;
2078         assert(t * val == 1);
2079         return t;
2080     }
2081
2082     /**
2083      * Calculate the multiplicative inverse of  $2^k \bmod \text{mod}$ , where mod is odd.
2084      */
2085     static MutableBigInteger modInverseBP2(MutableBigInteger mod, int k) {
2086         // Copy the mod to protect original
2087         return fixup(new MutableBigInteger(1), new MutableBigInteger(mod), k);
2088     }
2089
2090     /**
2091      * Calculate the multiplicative inverse of this mod mod, where mod is odd.
2092      * This and mod are not changed by the calculation.
2093      *
2094      * This method implements an algorithm due to Richard Schroepel, that uses
2095      * the same intermediate representation as Montgomery Reduction
2096      * ("Montgomery Form"). The algorithm is described in an unpublished
2097      * manuscript entitled "Fast Modular Reciprocals."
2098      */
2099     private MutableBigInteger modInverse(MutableBigInteger mod) {
2100         MutableBigInteger p = new MutableBigInteger(mod);
2101         MutableBigInteger f = new MutableBigInteger(this);
2102         MutableBigInteger g = new MutableBigInteger(p);
2103         SignedMutableBigInteger c = new SignedMutableBigInteger(1);
2104         SignedMutableBigInteger d = new SignedMutableBigInteger();
2105         MutableBigInteger temp = null;
2106         SignedMutableBigInteger sTemp = null;
2107
2108         int k = 0;
2109         // Right shift f k times until odd, left shift d k times
2110         if (f.isEven()) {
2111             int trailingZeros = f.getLowestSetBit();
2112             f.rightShift(trailingZeros);
2113             d.leftShift(trailingZeros);
2114             k = trailingZeros;
2115         }
2116
2117         // The Almost Inverse Algorithm
2118         while (!f.isOne()) {
2119             // If gcd(f, g) != 1, number is not invertible modulo mod
2120             if (f.isZero())
2121                 throw new ArithmeticException("BigInteger not invertible.");
2122
2123             // If f < g exchange f, g and c, d
2124             if (f.compare(g) < 0) {
2125                 temp = f; f = g; g = temp;
2126                 sTemp = d; d = c; c = sTemp;
2127             }
2128

```

```

2129         // If f == g (mod 4)
2130         if (((f.value[f.offset + f.intLen - 1] ^
2131             g.value[g.offset + g.intLen - 1]) & 3) == 0) {
2132             f.subtract(g);
2133             c.signedSubtract(d);
2134         } else { // If f != g (mod 4)
2135             f.add(g);
2136             c.signedAdd(d);
2137         }
2138
2139         // Right shift f k times until odd, left shift d k times
2140         int trailingZeros = f.getLowestSetBit();
2141         f.rightShift(trailingZeros);
2142         d.leftShift(trailingZeros);
2143         k += trailingZeros;
2144     }
2145
2146     while (c.sign < 0)
2147         c.signedAdd(p);
2148
2149     return fixup(c, p, k);
2150 }
2151
2152 /**
2153  * The Fixup Algorithm
2154  * Calculates X such that  $X = C * 2^{(-k)} \pmod{P}$ 
2155  * Assumes  $C < P$  and P is odd.
2156  */
2157 static MutableBigInteger fixup(MutableBigInteger c, MutableBigInteger p,
2158                               int k) {
2159     MutableBigInteger temp = new MutableBigInteger();
2160     // Set r to the multiplicative inverse of p mod  $2^{32}$ 
2161     int r = -inverseMod32(p.value[p.offset+p.intLen-1]);
2162
2163     for (int i=0, numWords = k >> 5; i < numWords; i++) {
2164         //  $V = R * c \pmod{2^j}$ 
2165         int v = r * c.value[c.offset + c.intLen-1];
2166         //  $c = c + (v * p)$ 
2167         p.mul(v, temp);
2168         c.add(temp);
2169         //  $c = c / 2^j$ 
2170         c.intLen--;
2171     }
2172     int numBits = k & 0x1f;
2173     if (numBits != 0) {
2174         //  $V = R * c \pmod{2^j}$ 
2175         int v = r * c.value[c.offset + c.intLen-1];
2176         v &= ((1<<numBits) - 1);
2177         //  $c = c + (v * p)$ 
2178         p.mul(v, temp);
2179         c.add(temp);
2180         //  $c = c / 2^j$ 
2181         c.rightShift(numBits);

```

```

2182     }
2183
2184     // In theory, c may be greater than p at this point (Very rare!)
2185     while (c.compare(p) >= 0)
2186         c.subtract(p);
2187
2188     return c;
2189 }
2190
2191 /**
2192  * Uses the extended Euclidean algorithm to compute the modInverse of base
2193  * mod a modulus that is a power of 2. The modulus is 2^k.
2194  */
2195 MutableBigInteger euclidModInverse(int k) {
2196     MutableBigInteger b = new MutableBigInteger(1);
2197     b.leftShift(k);
2198     MutableBigInteger mod = new MutableBigInteger(b);
2199
2200     MutableBigInteger a = new MutableBigInteger(this);
2201     MutableBigInteger q = new MutableBigInteger();
2202     MutableBigInteger r = b.divide(a, q);
2203
2204     MutableBigInteger swapper = b;
2205     // swap b & r
2206     b = r;
2207     r = swapper;
2208
2209     MutableBigInteger t1 = new MutableBigInteger(q);
2210     MutableBigInteger t0 = new MutableBigInteger(1);
2211     MutableBigInteger temp = new MutableBigInteger();
2212
2213     while (!b.isOne()) {
2214         r = a.divide(b, q);
2215
2216         if (r.intLen == 0)
2217             throw new ArithmeticException("BigInteger not invertible.");
2218
2219         swapper = r;
2220         a = swapper;
2221
2222         if (q.intLen == 1)
2223             t1.mul(q.value[q.offset], temp);
2224         else
2225             q.multiply(t1, temp);
2226         swapper = q;
2227         q = temp;
2228         temp = swapper;
2229         t0.add(q);
2230
2231         if (a.isOne())
2232             return t0;
2233
2234         r = b.divide(a, q);

```



```

2235
2236         if (r.intLen == 0)
2237             throw new ArithmeticException("BigInteger not invertible.");
2238
2239         swapper = b;
2240         b = r;
2241
2242         if (q.intLen == 1)
2243             t0.mul(q.value[q.offset], temp);
2244         else
2245             q.multiply(t0, temp);
2246         swapper = q; q = temp; temp = swapper;
2247
2248         t1.add(q);
2249     }
2250     mod.subtract(t1);
2251     return mod;
2252 }
2253 }

```

1.6 RoundingMode.java

Secuencia 7: java/math/RoundingMode.java

```

1  /*
2  * Copyright (c) 2003, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * Portions Copyright IBM Corporation, 2001. All Rights Reserved.
28 */
29 package java.math;
30

```

```

31 /**
32  * Specifies a <i>rounding behavior</i> for numerical operations
33  * capable of discarding precision. Each rounding mode indicates how
34  * the least significant returned digit of a rounded result is to be
35  * calculated. If fewer digits are returned than the digits needed to
36  * represent the exact numerical result, the discarded digits will be
37  * referred to as the <i>discarded fraction</i> regardless the digits'
38  * contribution to the value of the number. In other words,
39  * considered as a numerical value, the discarded fraction could have
40  * an absolute value greater than one.
41  *
42  * <p>Each rounding mode description includes a table listing how
43  * different two-digit decimal values would round to a one digit
44  * decimal value under the rounding mode in question. The result
45  * column in the tables could be gotten by creating a
46  * {@code BigDecimal} number with the specified value, forming a
47  * {@link MathContext} object with the proper settings
48  * ({@code precision} set to {@code 1}, and the
49  * {@code roundingMode} set to the rounding mode in question), and
50  * calling {@link BigDecimal#round round} on this number with the
51  * proper {@code MathContext}. A summary table showing the results
52  * of these rounding operations for all rounding modes appears below.
53  *
54  * <table border>
55  * <caption><b>Summary of Rounding Operations Under Different Rounding Modes</b></caption>
56  * <tr><th></th><th colspan=8>Result of rounding input to one digit with the given
57  * <th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th>
58  * <tr valign=top>
59  * <th>Input Number</th> <th>{@code UP}</th>
60  * <th> <th>{@code DOWN}</th>
61  * <th> <th>{@code CEILING}</th>
62  * <th> <th>{@code FLOOR}</th>
63  * <th> <th>{@code
64  * HALF_UP}</th>
65  *
66  * <th>{@code HALF_DOWN}</th>
67  *
68  * <th>{@code HALF_EVEN}</th>
69  *
70  * <th>{@code UNNECESSARY}</th>
71  *
72  * <tr align=right><td>5.5</td> <td>6</td> <td>5</td> <td>6</td> <td>5</td> <td>6</td>
73  * <td>5</td> <td>6</td> <td>throw {@code ArithmeticException}</td>
74  * <tr align=right><td>2.5</td> <td>3</td> <td>2</td> <td>3</td> <td>2</td> <td>3</td>
75  * <td>2</td> <td>2</td> <td>throw {@code ArithmeticException}</td>
76  * <tr align=right><td>1.6</td> <td>2</td> <td>1</td> <td>2</td> <td>1</td> <td>2</td>
77  * <td>2</td> <td>2</td> <td>throw {@code ArithmeticException}</td>
78  * <tr align=right><td>1.1</td> <td>2</td> <td>1</td> <td>2</td> <td>1</td> <td>1</td>
79  * <td>1</td> <td>1</td> <td>throw {@code ArithmeticException}</td>
80  * <tr align=right><td>1.0</td> <td>1</td> <td>1</td> <td>1</td> <td>1</td> <td>1</td>

```

```

73      <td>1</td>      <td>1</td>      <td>1</td>
74 * <tr align=right><td>-1.0</td> <td>-1</td> <td>-1</td> <td>-1</td> <td>-1</td> <td>-1</td>
75      <td>-1</td>      <td>-1</td>      <td>-1</td>
76 * <tr align=right><td>-1.1</td> <td>-2</td> <td>-1</td> <td>-1</td> <td>-2</td> <td>-1</td>
77      <td>-1</td>      <td>-1</td>      <td>throw {@code ArithmeticException}</td>
78 * <tr align=right><td>-1.6</td> <td>-2</td> <td>-1</td> <td>-1</td> <td>-2</td> <td>-2</td>
79      <td>-2</td>      <td>-2</td>      <td>throw {@code ArithmeticException}</td>
80 * <tr align=right><td>-2.5</td> <td>-3</td> <td>-2</td> <td>-2</td> <td>-3</td> <td>-3</td>
81      <td>-2</td>      <td>-2</td>      <td>throw {@code ArithmeticException}</td>
82 * <tr align=right><td>-5.5</td> <td>-6</td> <td>-5</td> <td>-5</td> <td>-6</td> <td>-6</td>
83      <td>-5</td>      <td>-6</td>      <td>throw {@code ArithmeticException}</td>
84 *</table>
85 *
86 * <p>This {@code enum} is intended to replace the integer-based
87 * enumeration of rounding mode constants in {@link BigDecimal}
88 * ({@link BigDecimal#ROUND_UP}, {@link BigDecimal#ROUND_DOWN},
89 * etc. ).
90 *
91 * @see    BigDecimal
92 * @see    MathContext
93 * @author  Josh Bloch
94 * @author  Mike Cowlishaw
95 * @author  Joseph D. Darcy
96 * @since  1.5
97 */
98 public enum RoundingMode {
99
100     /**
101      * Rounding mode to round away from zero. Always increments the
102      * digit prior to a non-zero discarded fraction. Note that this
103      * rounding mode never decreases the magnitude of the calculated
104      * value.
105      *
106      * <p>Example:
107      * <table border>
108      * <caption><b>Rounding mode UP Examples</b></caption>
109      * <tr valign=top><th>Input Number</th>
110      * <th>Input rounded to one digit<br> with {@code UP} rounding
111      * <tr align=right><td>5.5</td> <td>6</td>
112      * <tr align=right><td>2.5</td> <td>3</td>
113      * <tr align=right><td>1.6</td> <td>2</td>
114      * <tr align=right><td>1.1</td> <td>2</td>
115      * <tr align=right><td>1.0</td> <td>1</td>
116      * <tr align=right><td>-1.0</td> <td>-1</td>
117      * <tr align=right><td>-1.1</td> <td>-2</td>
118      * <tr align=right><td>-1.6</td> <td>-2</td>
119      * <tr align=right><td>-2.5</td> <td>-3</td>
120      * <tr align=right><td>-5.5</td> <td>-6</td>
121      * </table>
122      */
123     UP(BigDecimal.ROUND_UP),

```

```

120  /**
121   * Rounding mode to round towards zero. Never increments the digit
122   * prior to a discarded fraction (i.e., truncates). Note that this
123   * rounding mode never increases the magnitude of the calculated value.
124   *
125   * <p>Example:
126   * <table border>
127   *   <caption><b>Rounding mode DOWN Examples</b></caption>
128   *   <tr valign=top><th>Input Number</th>
129   *     <th>Input rounded to one digit<br> with {@code DOWN} rounding
130   *   <tr align=right><td>5.5</td> <td>5</td>
131   *   <tr align=right><td>2.5</td> <td>2</td>
132   *   <tr align=right><td>1.6</td> <td>1</td>
133   *   <tr align=right><td>1.1</td> <td>1</td>
134   *   <tr align=right><td>1.0</td> <td>1</td>
135   *   <tr align=right><td>-1.0</td> <td>-1</td>
136   *   <tr align=right><td>-1.1</td> <td>-1</td>
137   *   <tr align=right><td>-1.6</td> <td>-1</td>
138   *   <tr align=right><td>-2.5</td> <td>-2</td>
139   *   <tr align=right><td>-5.5</td> <td>-5</td>
140   * </table>
141   */
142  DOWN(BigDecimal.ROUND_DOWN),
143
144  /**
145   * Rounding mode to round towards positive infinity. If the
146   * result is positive, behaves as for {@code RoundingMode.UP};
147   * if negative, behaves as for {@code RoundingMode.DOWN}. Note
148   * that this rounding mode never decreases the calculated value.
149   *
150   * <p>Example:
151   * <table border>
152   *   <caption><b>Rounding mode CEILING Examples</b></caption>
153   *   <tr valign=top><th>Input Number</th>
154   *     <th>Input rounded to one digit<br> with {@code CEILING} rounding
155   *   <tr align=right><td>5.5</td> <td>6</td>
156   *   <tr align=right><td>2.5</td> <td>3</td>
157   *   <tr align=right><td>1.6</td> <td>2</td>
158   *   <tr align=right><td>1.1</td> <td>2</td>
159   *   <tr align=right><td>1.0</td> <td>1</td>
160   *   <tr align=right><td>-1.0</td> <td>-1</td>
161   *   <tr align=right><td>-1.1</td> <td>-1</td>
162   *   <tr align=right><td>-1.6</td> <td>-1</td>
163   *   <tr align=right><td>-2.5</td> <td>-2</td>
164   *   <tr align=right><td>-5.5</td> <td>-5</td>
165   * </table>
166   */
167  CEILING(BigDecimal.ROUND_CEILING),
168
169  /**
170   * Rounding mode to round towards negative infinity. If the
171   * result is positive, behave as for {@code RoundingMode.DOWN};
172   * if negative, behave as for {@code RoundingMode.UP}. Note that

```

```

173      * this rounding mode never increases the calculated value.
174      *
175      *<p>Example:
176      *<table border>
177      * <caption><b>Rounding mode FLOOR Examples</b></caption>
178      *<tr valign=top><th>Input Number</th>
179      *   <th>Input rounded to one digit<br> with {@code FLOOR} rounding
180      *<tr align=right><td>5.5</td> <td>5</td>
181      *<tr align=right><td>2.5</td> <td>2</td>
182      *<tr align=right><td>1.6</td> <td>1</td>
183      *<tr align=right><td>1.1</td> <td>1</td>
184      *<tr align=right><td>1.0</td> <td>1</td>
185      *<tr align=right><td>-1.0</td> <td>-1</td>
186      *<tr align=right><td>-1.1</td> <td>-2</td>
187      *<tr align=right><td>-1.6</td> <td>-2</td>
188      *<tr align=right><td>-2.5</td> <td>-3</td>
189      *<tr align=right><td>-5.5</td> <td>-6</td>
190      *</table>
191      */
192      FLOOR(BigDecimal.ROUND_FLOOR),
193
194      /**
195       * Rounding mode to round towards {@literal "nearest neighbor"}
196       * unless both neighbors are equidistant, in which case round up.
197       * Behaves as for {@code RoundingMode.UP} if the discarded
198       * fraction is  $\geq 0.5$ ; otherwise, behaves as for
199       * {@code RoundingMode.DOWN}. Note that this is the rounding
200       * mode commonly taught at school.
201       *
202       *<p>Example:
203       *<table border>
204       * <caption><b>Rounding mode HALF_UP Examples</b></caption>
205       *<tr valign=top><th>Input Number</th>
206       *   <th>Input rounded to one digit<br> with {@code HALF_UP} rounding
207       *<tr align=right><td>5.5</td> <td>6</td>
208       *<tr align=right><td>2.5</td> <td>3</td>
209       *<tr align=right><td>1.6</td> <td>2</td>
210       *<tr align=right><td>1.1</td> <td>1</td>
211       *<tr align=right><td>1.0</td> <td>1</td>
212       *<tr align=right><td>-1.0</td> <td>-1</td>
213       *<tr align=right><td>-1.1</td> <td>-1</td>
214       *<tr align=right><td>-1.6</td> <td>-2</td>
215       *<tr align=right><td>-2.5</td> <td>-3</td>
216       *<tr align=right><td>-5.5</td> <td>-6</td>
217       *</table>
218       */
219      HALF_UP(BigDecimal.ROUND_HALF_UP),
220
221      /**
222       * Rounding mode to round towards {@literal "nearest neighbor"}
223       * unless both neighbors are equidistant, in which case round
224       * down. Behaves as for {@code RoundingMode.UP} if the discarded
225       * fraction is  $\geq 0.5$ ; otherwise, behaves as for

```

```

226      * {@code RoundingMode.DOWN}.
227      *
228      *<p>Example:
229      *<table border>
230      * <caption><b>Rounding mode HALF_DOWN Examples</b></caption>
231      *<tr valign=top><th>Input Number</th>
232      *   <th>Input rounded to one digit<br> with {@code HALF_DOWN} rounding
233      *<tr align=right><td>5.5</td> <td>5</td>
234      *<tr align=right><td>2.5</td> <td>2</td>
235      *<tr align=right><td>1.6</td> <td>2</td>
236      *<tr align=right><td>1.1</td> <td>1</td>
237      *<tr align=right><td>1.0</td> <td>1</td>
238      *<tr align=right><td>-1.0</td> <td>-1</td>
239      *<tr align=right><td>-1.1</td> <td>-1</td>
240      *<tr align=right><td>-1.6</td> <td>-2</td>
241      *<tr align=right><td>-2.5</td> <td>-2</td>
242      *<tr align=right><td>-5.5</td> <td>-5</td>
243      *</table>
244      */
245      HALF_DOWN(BigDecimal.ROUND_HALF_DOWN),
246
247      /**
248       * Rounding mode to round towards the {@literal "nearest neighbor"}
249       * unless both neighbors are equidistant, in which case, round
250       * towards the even neighbor. Behaves as for
251       * {@code RoundingMode.HALF_UP} if the digit to the left of the
252       * discarded fraction is odd; behaves as for
253       * {@code RoundingMode.HALF_DOWN} if it's even. Note that this
254       * is the rounding mode that statistically minimizes cumulative
255       * error when applied repeatedly over a sequence of calculations.
256       * It is sometimes known as {@literal "Banker's rounding,"} and is
257       * chiefly used in the USA. This rounding mode is analogous to
258       * the rounding policy used for {@code float} and {@code double}
259       * arithmetic in Java.
260       *
261       *<p>Example:
262       *<table border>
263       * <caption><b>Rounding mode HALF_EVEN Examples</b></caption>
264       *<tr valign=top><th>Input Number</th>
265       *   <th>Input rounded to one digit<br> with {@code HALF_EVEN} rounding
266       *<tr align=right><td>5.5</td> <td>6</td>
267       *<tr align=right><td>2.5</td> <td>2</td>
268       *<tr align=right><td>1.6</td> <td>2</td>
269       *<tr align=right><td>1.1</td> <td>1</td>
270       *<tr align=right><td>1.0</td> <td>1</td>
271       *<tr align=right><td>-1.0</td> <td>-1</td>
272       *<tr align=right><td>-1.1</td> <td>-1</td>
273       *<tr align=right><td>-1.6</td> <td>-2</td>
274       *<tr align=right><td>-2.5</td> <td>-2</td>
275       *<tr align=right><td>-5.5</td> <td>-6</td>
276       *</table>
277       */
278      HALF_EVEN(BigDecimal.ROUND_HALF_EVEN),

```

```

279
280 /**
281  * Rounding mode to assert that the requested operation has an exact
282  * result, hence no rounding is necessary. If this rounding mode is
283  * specified on an operation that yields an inexact result, an
284  * {@code ArithmeticException} is thrown.
285  * <p>Example:
286  * <table border>
287  *   <caption><b>Rounding mode UNNECESSARY Examples</b></caption>
288  *   <tr valign=top><th>Input Number</th>
289  *     <th>Input rounded to one digit<br> with {@code UNNECESSARY} rounding
290  *   <tr align=right><td>5.5</td> <td>throw {@code ArithmeticException}</td>
291  *   <tr align=right><td>2.5</td> <td>throw {@code ArithmeticException}</td>
292  *   <tr align=right><td>1.6</td> <td>throw {@code ArithmeticException}</td>
293  *   <tr align=right><td>1.1</td> <td>throw {@code ArithmeticException}</td>
294  *   <tr align=right><td>1.0</td> <td>1</td>
295  *   <tr align=right><td>-1.0</td> <td>-1</td>
296  *   <tr align=right><td>-1.1</td> <td>throw {@code ArithmeticException}</td>
297  *   <tr align=right><td>-1.6</td> <td>throw {@code ArithmeticException}</td>
298  *   <tr align=right><td>-2.5</td> <td>throw {@code ArithmeticException}</td>
299  *   <tr align=right><td>-5.5</td> <td>throw {@code ArithmeticException}</td>
300  * </table>
301  */
302 UNNECESSARY(BigDecimal.ROUND_UNNECESSARY);
303
304 // Corresponding BigDecimal rounding constant
305 final int oldMode;
306
307 /**
308  * Constructor
309  *
310  * @param oldMode The {@code BigDecimal} constant corresponding to
311  *   this mode
312  */
313 private RoundingMode(int oldMode) {
314     this.oldMode = oldMode;
315 }
316
317 /**
318  * Returns the {@code RoundingMode} object corresponding to a
319  * legacy integer rounding mode constant in {@link BigDecimal}.
320  *
321  * @param rm legacy integer rounding mode to convert
322  * @return {@code RoundingMode} corresponding to the given integer.
323  * @throws IllegalArgumentException integer is out of range
324  */
325 public static RoundingMode valueOf(int rm) {
326     switch(rm) {
327
328     case BigDecimal.ROUND_UP:
329         return UP;
330
331     case BigDecimal.ROUND_DOWN:

```

```

332         return DOWN;
333
334     case BigDecimal.ROUND_CEILING:
335         return CEILING;
336
337     case BigDecimal.ROUND_FLOOR:
338         return FLOOR;
339
340     case BigDecimal.ROUND_HALF_UP:
341         return HALF_UP;
342
343     case BigDecimal.ROUND_HALF_DOWN:
344         return HALF_DOWN;
345
346     case BigDecimal.ROUND_HALF_EVEN:
347         return HALF_EVEN;
348
349     case BigDecimal.ROUND_UNNECESSARY:
350         return UNNECESSARY;
351
352     default:
353         throw new IllegalArgumentException("argument out of range");
354     }
355 }
356 }

```

1.7 SignedMutableBigInteger.java

Secuencia 8: java/math/SignedMutableBigInteger.java

```

1  /*
2  * Copyright (c) 1999, 2007, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */

```



```
25
26 package java.math;
27
28 /**
29  * A class used to represent multiprecision integers that makes efficient
30  * use of allocated space by allowing a number to occupy only part of
31  * an array so that the arrays do not have to be reallocated as often.
32  * When performing an operation with many iterations the array used to
33  * hold a number is only increased when necessary and does not have to
34  * be the same size as the number it represents. A mutable number allows
35  * calculations to occur on the same number without having to create
36  * a new number for every step of the calculation as occurs with
37  * BigIntegers.
38  *
39  * Note that SignedMutableBigIntegers only support signed addition and
40  * subtraction. All other operations occur as with MutableBigIntegers.
41  *
42  * @see      BigInteger
43  * @author    Michael McCloskey
44  * @since     1.3
45  */
46
47 class SignedMutableBigInteger extends MutableBigInteger {
48
49     /**
50      * The sign of this MutableBigInteger.
51      */
52     int sign = 1;
53
54     // Constructors
55
56     /**
57      * The default constructor. An empty MutableBigInteger is created with
58      * a one word capacity.
59      */
60     SignedMutableBigInteger() {
61         super();
62     }
63
64     /**
65      * Construct a new MutableBigInteger with a magnitude specified by
66      * the int val.
67      */
68     SignedMutableBigInteger(int val) {
69         super(val);
70     }
71
72     /**
73      * Construct a new MutableBigInteger with a magnitude equal to the
74      * specified MutableBigInteger.
75      */
76     SignedMutableBigInteger(MutableBigInteger val) {
77         super(val);
```

```
78     }
79
80     // Arithmetic Operations
81
82     /**
83      * Signed addition built upon unsigned add and subtract.
84      */
85     void signedAdd(SignedMutableBigInteger addend) {
86         if (sign == addend.sign)
87             add(addend);
88         else
89             sign = sign * subtract(addend);
90     }
91
92
93     /**
94      * Signed addition built upon unsigned add and subtract.
95      */
96     void signedAdd(MutableBigInteger addend) {
97         if (sign == 1)
98             add(addend);
99         else
100             sign = sign * subtract(addend);
101     }
102
103
104     /**
105      * Signed subtraction built upon unsigned add and subtract.
106      */
107     void signedSubtract(SignedMutableBigInteger addend) {
108         if (sign == addend.sign)
109             sign = sign * subtract(addend);
110         else
111             add(addend);
112     }
113
114
115     /**
116      * Signed subtraction built upon unsigned add and subtract.
117      */
118     void signedSubtract(MutableBigInteger addend) {
119         if (sign == 1)
120             sign = sign * subtract(addend);
121         else
122             add(addend);
123         if (intLen == 0)
124             sign = 1;
125     }
126
127     /**
128      * Print out the first intLen ints of this MutableBigInteger's value
129      * array starting at offset.
130      */
```

```
131     public String toString() {
132         return this.toBigInteger(sign).toString();
133     }
134
135 }
```

1.8 package-info.java

Secuencia 9: java/math/package-info.java

```
1  /*
2  * Copyright (c) 1998, 2006, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /**
27  * Provides classes for performing arbitrary-precision integer
28  * arithmetic ({@code BigInteger}) and arbitrary-precision decimal
29  * arithmetic ({@code BigDecimal}).  {@code BigInteger} is analogous
30  * to the primitive integer types except that it provides arbitrary
31  * precision, hence operations on {@code BigInteger}s do not overflow
32  * or lose precision.  In addition to standard arithmetic operations,
33  * {@code BigInteger} provides modular arithmetic, GCD calculation,
34  * primality testing, prime generation, bit manipulation, and a few
35  * other miscellaneous operations.
36  *
37  * {@code BigDecimal} provides arbitrary-precision signed decimal
38  * numbers suitable for currency calculations and the like.  {@code
39  * BigDecimal} gives the user complete control over rounding behavior,
40  * allowing the user to choose from a comprehensive set of eight
41  * rounding modes.
42  *
43  * @since JDK1.1
44  */
```

```
45 package java.math;
```

2 Text (java.text package)

2.1 Annotation.java

Secuencia 10: java/text/Annotation.java

```
1  /*
2  * Copyright (c) 1997, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 package java.text;
27
28 /**
29 * An Annotation object is used as a wrapper for a text attribute value if
30 * the attribute has annotation characteristics. These characteristics are:
31 * <ul>
32 * <li>The text range that the attribute is applied to is critical to the
33 * semantics of the range. That means, the attribute cannot be applied to subranges
34 * of the text range that it applies to, and, if two adjacent text ranges have
35 * the same value for this attribute, the attribute still cannot be applied to
36 * the combined range as a whole with this value.
37 * <li>The attribute or its value usually do no longer apply if the underlying text is
38 * changed.
39 * </ul>
40 *
41 * An example is grammatical information attached to a sentence:
42 * For the previous sentence, you can say that "an example"
43 * is the subject, but you cannot say the same about "an", "example", or "exam".
44 * When the text is changed, the grammatical information typically becomes invalid.
45 * Another example is Japanese reading information (yomi).
46 *
```

```

47 * <p>
48 * Wrapping the attribute value into an Annotation object guarantees that
49 * adjacent text runs don't get merged even if the attribute values are equal,
50 * and indicates to text containers that the attribute should be discarded if
51 * the underlying text is modified.
52 *
53 * @see AttributedCharacterIterator
54 * @since 1.2
55 */
56
57 public class Annotation {
58
59     /**
60      * Constructs an annotation record with the given value, which
61      * may be null.
62      *
63      * @param value the value of the attribute
64      */
65     public Annotation(Object value) {
66         this.value = value;
67     }
68
69     /**
70      * Returns the value of the attribute, which may be null.
71      *
72      * @return the value of the attribute
73      */
74     public Object getValue() {
75         return value;
76     }
77
78     /**
79      * Returns the String representation of this Annotation.
80      *
81      * @return the {@code String} representation of this {@code Annotation}
82      */
83     public String toString() {
84         return getClass().getName() + "[value=" + value + "]";
85     }
86
87     private Object value;
88
89 };

```

2.2 AttributedCharacterIterator.java

Secuencia 11: java/text/AttributedCharacterIterator.java

```

1 /*
2  * Copyright (c) 1997, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *

```

```
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 package java.text;
27
28 import java.io.InvalidObjectException;
29 import java.io.Serializable;
30 import java.util.HashMap;
31 import java.util.Map;
32 import java.util.Set;
33
34 /**
35  * An {@code AttributedCharacterIterator} allows iteration through both text and
36  * related attribute information.
37  *
38  * <p>
39  * An attribute is a key/value pair, identified by the key. No two
40  * attributes on a given character can have the same key.
41  *
42  * <p>The values for an attribute are immutable, or must not be mutated
43  * by clients or storage. They are always passed by reference, and not
44  * cloned.
45  *
46  * <p>A <em>run with respect to an attribute</em> is a maximum text range for
47  * which:
48  * <ul>
49  * <li>the attribute is undefined or {@code null} for the entire range, or
50  * <li>the attribute value is defined and has the same non-{@code null} value for the
51  *     entire range.
52  * </ul>
53  *
54  * <p>A <em>run with respect to a set of attributes</em> is a maximum text range for
55  * which this condition is met for each member attribute.
56  *
57  * <p>When getting a run with no explicit attributes specified (i.e.,
58  * calling {@link #getRunStart()} and {@link #getRunLimit()}), any
59  * contiguous text segments having the same attributes (the same set
```

```

60  * of attribute/value pairs) are treated as separate runs if the
61  * attributes have been given to those text segments separately.
62  *
63  * <p>The returned indexes are limited to the range of the iterator.
64  *
65  * <p>The returned attribute information is limited to runs that contain
66  * the current character.
67  *
68  * <p>
69  * Attribute keys are instances of {@link AttributedCharacterIterator.Attribute} and its
70  * subclasses, such as {@link java.awt.font.TextAttribute}.
71  *
72  * @see AttributedCharacterIterator.Attribute
73  * @see java.awt.font.TextAttribute
74  * @see AttributedString
75  * @see Annotation
76  * @since 1.2
77  */
78
79  public interface AttributedCharacterIterator extends CharacterIterator {
80
81      /**
82       * Defines attribute keys that are used to identify text attributes. These
83       * keys are used in {@code AttributedCharacterIterator} and {@code AttributedString}.
84       * @see AttributedCharacterIterator
85       * @see AttributedString
86       * @since 1.2
87       */
88
89      public static class Attribute implements Serializable {
90
91          /**
92           * The name of this {@code Attribute}. The name is used primarily by {@code readResolve}
93           * to look up the corresponding predefined instance when deserializing
94           * an instance.
95           * @serial
96           */
97          private String name;
98
99          // table of all instances in this class, used by readResolve
100         private static final Map<String, Attribute> instanceMap = new HashMap<>(7);
101
102         /**
103          * Constructs an {@code Attribute} with the given name.
104          *
105          * @param name the name of {@code Attribute}
106          */
107         protected Attribute(String name) {
108             this.name = name;
109             if (this.getClass() == Attribute.class) {
110                 instanceMap.put(name, this);
111             }
112         }

```

```

113
114     /**
115      * Compares two objects for equality. This version only returns true
116      * for {@code x.equals(y)} if {@code x} and {@code y} refer
117      * to the same object, and guarantees this for all subclasses.
118      */
119     public final boolean equals(Object obj) {
120         return super.equals(obj);
121     }
122
123     /**
124      * Returns a hash code value for the object. This version is identical to
125      * the one in {@code Object}, but is also final.
126      */
127     public final int hashCode() {
128         return super.hashCode();
129     }
130
131     /**
132      * Returns a string representation of the object. This version returns the
133      * concatenation of class name, {@code "("}, a name identifying the attribute
134      * and {@code ")"}.
135      */
136     public String toString() {
137         return getClass().getName() + "(" + name + ")";
138     }
139
140     /**
141      * Returns the name of the attribute.
142      *
143      * @return the name of {@code Attribute}
144      */
145     protected String getName() {
146         return name;
147     }
148
149     /**
150      * Resolves instances being deserialized to the predefined constants.
151      *
152      * @return the resolved {@code Attribute} object
153      * @throws InvalidObjectException if the object to resolve is not
154      *         an instance of {@code Attribute}
155      */
156     protected Object readResolve() throws InvalidObjectException {
157         if (this.getClass() != Attribute.class) {
158             throw new InvalidObjectException("subclass didn't correctly implement readResolve")
159                 ;
160         }
161
162         Attribute instance = instanceMap.get(getName());
163         if (instance != null) {
164             return instance;
165         } else {

```



```

165         throw new InvalidObjectException("unknown attribute name");
166     }
167 }
168
169 /**
170  * Attribute key for the language of some text.
171  * <p> Values are instances of {@link java.util.Locale Locale}.
172  * @see java.util.Locale
173  */
174 public static final Attribute LANGUAGE = new Attribute("language");
175
176 /**
177  * Attribute key for the reading of some text. In languages where the written form
178  * and the pronunciation of a word are only loosely related (such as Japanese),
179  * it is often necessary to store the reading (pronunciation) along with the
180  * written form.
181  * <p>Values are instances of {@link Annotation} holding instances of {@link String}.
182  *
183  * @see Annotation
184  * @see java.lang.String
185  */
186 public static final Attribute READING = new Attribute("reading");
187
188 /**
189  * Attribute key for input method segments. Input methods often break
190  * up text into segments, which usually correspond to words.
191  * <p>Values are instances of {@link Annotation} holding a {@code null} reference.
192  * @see Annotation
193  */
194 public static final Attribute INPUT_METHOD_SEGMENT = new Attribute("input_method_segment");
195
196 // make sure the serial version doesn't change between compiler versions
197 private static final long serialVersionUID = -9142742483513960612L;
198
199 };
200
201 /**
202  * Returns the index of the first character of the run
203  * with respect to all attributes containing the current character.
204  *
205  * <p>Any contiguous text segments having the same attributes (the
206  * same set of attribute/value pairs) are treated as separate runs
207  * if the attributes have been given to those text segments separately.
208  *
209  * @return the index of the first character of the run
210  */
211 public int getRunStart();
212
213 /**
214  * Returns the index of the first character of the run
215  * with respect to the given {@code attribute} containing the current character.
216  *
217  * @param attribute the desired attribute.

```

```
218     * @return the index of the first character of the run
219     */
220     public int getRunStart(Attribute attribute);
221
222     /**
223     * Returns the index of the first character of the run
224     * with respect to the given {@code attributes} containing the current character.
225     *
226     * @param attributes a set of the desired attributes.
227     * @return the index of the first character of the run
228     */
229     public int getRunStart(Set<? extends Attribute> attributes);
230
231     /**
232     * Returns the index of the first character following the run
233     * with respect to all attributes containing the current character.
234     *
235     * <p>Any contiguous text segments having the same attributes (the
236     * same set of attribute/value pairs) are treated as separate runs
237     * if the attributes have been given to those text segments separately.
238     *
239     * @return the index of the first character following the run
240     */
241     public int getRunLimit();
242
243     /**
244     * Returns the index of the first character following the run
245     * with respect to the given {@code attribute} containing the current character.
246     *
247     * @param attribute the desired attribute
248     * @return the index of the first character following the run
249     */
250     public int getRunLimit(Attribute attribute);
251
252     /**
253     * Returns the index of the first character following the run
254     * with respect to the given {@code attributes} containing the current character.
255     *
256     * @param attributes a set of the desired attributes
257     * @return the index of the first character following the run
258     */
259     public int getRunLimit(Set<? extends Attribute> attributes);
260
261     /**
262     * Returns a map with the attributes defined on the current
263     * character.
264     *
265     * @return a map with the attributes defined on the current character
266     */
267     public Map<Attribute, Object> getAttributes();
268
269     /**
270     * Returns the value of the named {@code attribute} for the current character.
```

```

271     * Returns {@code null} if the {@code attribute} is not defined.
272     *
273     * @param attribute the desired attribute
274     * @return the value of the named {@code attribute} or {@code null}
275     */
276     public Object getAttribute(Attribute attribute);
277
278     /**
279     * Returns the keys of all attributes defined on the
280     * iterator's text range. The set is empty if no
281     * attributes are defined.
282     *
283     * @return the keys of all attributes
284     */
285     public Set<Attribute> getAllAttributeKeys();
286 };

```

2.3 AttributedString.java

Secuencia 12: java/text/AttributedString.java

```

1  /*
2  * Copyright (c) 1997, 2012, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 package java.text;
27
28 import java.util.*;
29 import java.text.AttributedString.Attribute;
30
31 /**
32 * An AttributedString holds text and related attribute information. It
33 * may be used as the actual data storage in some cases where a text

```

```

34 * reader wants to access attributed text through the AttributedStringCharacterIterator
35 * interface.
36 *
37 * <p>
38 * An attribute is a key/value pair, identified by the key. No two
39 * attributes on a given character can have the same key.
40 *
41 * <p>The values for an attribute are immutable, or must not be mutated
42 * by clients or storage. They are always passed by reference, and not
43 * cloned.
44 *
45 * @see AttributedStringCharacterIterator
46 * @see Annotation
47 * @since 1.2
48 */
49
50 public class AttributedString {
51
52     // since there are no vectors of int, we have to use arrays.
53     // We allocate them in chunks of 10 elements so we don't have to allocate all the time.
54     private static final int ARRAY_SIZE_INCREMENT = 10;
55
56     // field holding the text
57     String text;
58
59     // fields holding run attribute information
60     // run attributes are organized by run
61     int runArraySize;           // current size of the arrays
62     int runCount;               // actual number of runs, <= runArraySize
63     int runStarts[];           // start index for each run
64     Vector<Attribute> runAttributes[]; // vector of attribute keys for each run
65     Vector<Object> runAttributeValues[]; // parallel vector of attribute values for each run
66
67     /**
68      * Constructs an AttributedString instance with the given
69      * AttributedStringCharacterIterators.
70      *
71      * @param iterators AttributedStringCharacterIterators to construct
72      * AttributedString from.
73      * @throws NullPointerException if iterators is null
74      */
75     AttributedString(AttributedStringCharacterIterator[] iterators) {
76         if (iterators == null) {
77             throw new NullPointerException("Iterators must not be null");
78         }
79         if (iterators.length == 0) {
80             text = "";
81         }
82         else {
83             // Build the String contents
84             StringBuffer buffer = new StringBuffer();
85             for (int counter = 0; counter < iterators.length; counter++) {
86                 appendContents(buffer, iterators[counter]);
87             }
88         }
89     }
90
91     /**
92      * Appends the contents of the given AttributedStringCharacterIterator
93      * to the end of the text.
94      *
95      * @param iterator AttributedStringCharacterIterator to append
96      */
97     void appendContents(StringBuffer buffer, AttributedStringCharacterIterator iterator) {
98         // Append the text
99         buffer.append(iterator.getText());
100
101         // Append the attributes
102         if (iterator.hasAttributes()) {
103             Vector<Attribute> attributes = iterator.getAttributes();
104             for (int i = 0; i < attributes.size(); i++) {
105                 Attribute attribute = attributes.get(i);
106                 Object value = attribute.getValue();
107                 buffer.append(attribute.getKey().toString() + "=" + value.toString() + " ");
108             }
109         }
110     }
111
112     /**
113      * Returns the text of this AttributedString.
114      *
115      * @return String text
116      */
117     String getText() {
118         return text;
119     }
120
121     /**
122      * Returns the attributes of this AttributedString.
123      *
124      * @return Vector<Attribute> attributes
125      */
126     Vector<Attribute> getAttributes() {
127         return null;
128     }
129
130     /**
131      * Returns the character at the given index.
132      *
133      * @param index index of character
134      * @return char character
135      */
136     char getChar(int index) {
137         return text.charAt(index);
138     }
139
140     /**
141      * Returns the length of the text.
142      *
143      * @return int length
144      */
145     int getLength() {
146         return text.length();
147     }
148
149     /**
150      * Returns the number of runs in this AttributedString.
151      *
152      * @return int runCount
153      */
154     int getRunCount() {
155         return runCount;
156     }
157
158     /**
159      * Returns the start index of the run at the given index.
160      *
161      * @param runIndex index of run
162      * @return int start index
163      */
164     int getRunStart(int runIndex) {
165         return runStarts[runIndex];
166     }
167
168     /**
169      * Returns the attribute keys of the run at the given index.
170      *
171      * @param runIndex index of run
172      * @return Vector<Attribute> attributes
173      */
174     Vector<Attribute> getRunAttributes(int runIndex) {
175         return runAttributes[runIndex];
176     }
177
178     /**
179      * Returns the attribute values of the run at the given index.
180      *
181      * @param runIndex index of run
182      * @return Vector<Object> values
183      */
184     Vector<Object> getRunAttributeValues(int runIndex) {
185         return runAttributeValues[runIndex];
186     }
187
188     /**
189      * Returns the character at the given index, including attributes.
190      *
191      * @param index index of character
192      * @return String character
193      */
194     String getCharWithAttributes(int index) {
195         return text.charAt(index) + " ";
196     }
197
198     /**
199      * Returns the length of the text, including attributes.
200      *
201      * @return int length
202      */
203     int getLengthWithAttributes() {
204         return text.length() + runCount * 2;
205     }
206
207     /**
208      * Returns the number of runs in this AttributedString, including attributes.
209      *
210      * @return int runCount
211      */
212     int getRunCountWithAttributes() {
213         return runCount;
214     }
215
216     /**
217      * Returns the start index of the run at the given index, including attributes.
218      *
219      * @param runIndex index of run
220      * @return int start index
221      */
222     int getRunStartWithAttributes(int runIndex) {
223         return runStarts[runIndex];
224     }
225
226     /**
227      * Returns the attribute keys of the run at the given index, including attributes.
228      *
229      * @param runIndex index of run
230      * @return Vector<Attribute> attributes
231      */
232     Vector<Attribute> getRunAttributesWithAttributes(int runIndex) {
233         return runAttributes[runIndex];
234     }
235
236     /**
237      * Returns the attribute values of the run at the given index, including attributes.
238      *
239      * @param runIndex index of run
240      * @return Vector<Object> values
241      */
242     Vector<Object> getRunAttributeValuesWithAttributes(int runIndex) {
243         return runAttributeValues[runIndex];
244     }
245
246     /**
247      * Returns the character at the given index, including attributes and styling.
248      *
249      * @param index index of character
250      * @return String character
251      */
252     String getCharWithAttributesAndStyling(int index) {
253         return text.charAt(index) + " ";
254     }
255
256     /**
257      * Returns the length of the text, including attributes and styling.
258      *
259      * @return int length
260      */
261     int getLengthWithAttributesAndStyling() {
262         return text.length() + runCount * 2;
263     }
264
265     /**
266      * Returns the number of runs in this AttributedString, including attributes and styling.
267      *
268      * @return int runCount
269      */
270     int getRunCountWithAttributesAndStyling() {
271         return runCount;
272     }
273
274     /**
275      * Returns the start index of the run at the given index, including attributes and styling.
276      *
277      * @param runIndex index of run
278      * @return int start index
279      */
280     int getRunStartWithAttributesAndStyling(int runIndex) {
281         return runStarts[runIndex];
282     }
283
284     /**
285      * Returns the attribute keys of the run at the given index, including attributes and styling.
286      *
287      * @param runIndex index of run
288      * @return Vector<Attribute> attributes
289      */
290     Vector<Attribute> getRunAttributesWithAttributesAndStyling(int runIndex) {
291         return runAttributes[runIndex];
292     }
293
294     /**
295      * Returns the attribute values of the run at the given index, including attributes and styling.
296      *
297      * @param runIndex index of run
298      * @return Vector<Object> values
299      */
300     Vector<Object> getRunAttributeValuesWithAttributesAndStyling(int runIndex) {
301         return runAttributeValues[runIndex];
302     }
303
304     /**
305      * Returns the character at the given index, including attributes, styling, and font.
306      *
307      * @param index index of character
308      * @return String character
309      */
310     String getCharWithAttributesAndStylingAndFont(int index) {
311         return text.charAt(index) + " ";
312     }
313
314     /**
315      * Returns the length of the text, including attributes, styling, and font.
316      *
317      * @return int length
318      */
319     int getLengthWithAttributesAndStylingAndFont() {
320         return text.length() + runCount * 2;
321     }
322
323     /**
324      * Returns the number of runs in this AttributedString, including attributes, styling, and font.
325      *
326      * @return int runCount
327      */
328     int getRunCountWithAttributesAndStylingAndFont() {
329         return runCount;
330     }
331
332     /**
333      * Returns the start index of the run at the given index, including attributes, styling, and font.
334      *
335      * @param runIndex index of run
336      * @return int start index
337      */
338     int getRunStartWithAttributesAndStylingAndFont(int runIndex) {
339         return runStarts[runIndex];
340     }
341
342     /**
343      * Returns the attribute keys of the run at the given index, including attributes, styling, and font.
344      *
345      * @param runIndex index of run
346      * @return Vector<Attribute> attributes
347      */
348     Vector<Attribute> getRunAttributesWithAttributesAndStylingAndFont(int runIndex) {
349         return runAttributes[runIndex];
350     }
351
352     /**
353      * Returns the attribute values of the run at the given index, including attributes, styling, and font.
354      *
355      * @param runIndex index of run
356      * @return Vector<Object> values
357      */
358     Vector<Object> getRunAttributeValuesWithAttributesAndStylingAndFont(int runIndex) {
359         return runAttributeValues[runIndex];
360     }
361
362     /**
363      * Returns the character at the given index, including attributes, styling, font, and color.
364      *
365      * @param index index of character
366      * @return String character
367      */
368     String getCharWithAttributesAndStylingAndFontAndColor(int index) {
369         return text.charAt(index) + " ";
370     }
371
372     /**
373      * Returns the length of the text, including attributes, styling, font, and color.
374      *
375      * @return int length
376      */
377     int getLengthWithAttributesAndStylingAndFontAndColor() {
378         return text.length() + runCount * 2;
379     }
380
381     /**
382      * Returns the number of runs in this AttributedString, including attributes, styling, font, and color.
383      *
384      * @return int runCount
385      */
386     int getRunCountWithAttributesAndStylingAndFontAndColor() {
387         return runCount;
388     }
389
390     /**
391      * Returns the start index of the run at the given index, including attributes, styling, font, and color.
392      *
393      * @param runIndex index of run
394      * @return int start index
395      */
396     int getRunStartWithAttributesAndStylingAndFontAndColor(int runIndex) {
397         return runStarts[runIndex];
398     }
399
400     /**
401      * Returns the attribute keys of the run at the given index, including attributes, styling, font, and color.
402      *
403      * @param runIndex index of run
404      * @return Vector<Attribute> attributes
405      */
406     Vector<Attribute> getRunAttributesWithAttributesAndStylingAndFontAndColor(int runIndex) {
407         return runAttributes[runIndex];
408     }
409
410     /**
411      * Returns the attribute values of the run at the given index, including attributes, styling, font, and color.
412      *
413      * @param runIndex index of run
414      * @return Vector<Object> values
415      */
416     Vector<Object> getRunAttributeValuesWithAttributesAndStylingAndFontAndColor(int runIndex) {
417         return runAttributeValues[runIndex];
418     }
419
420     /**
421      * Returns the character at the given index, including attributes, styling, font, color, and background color.
422      *
423      * @param index index of character
424      * @return String character
425      */
426     String getCharWithAttributesAndStylingAndFontAndColorAndBackgroundColor(int index) {
427         return text.charAt(index) + " ";
428     }
429
430     /**
431      * Returns the length of the text, including attributes, styling, font, color, and background color.
432      *
433      * @return int length
434      */
435     int getLengthWithAttributesAndStylingAndFontAndColorAndBackgroundColor() {
436         return text.length() + runCount * 2;
437     }
438
439     /**
440      * Returns the number of runs in this AttributedString, including attributes, styling, font, color, and background color.
441      *
442      * @return int runCount
443      */
444     int getRunCountWithAttributesAndStylingAndFontAndColorAndBackgroundColor() {
445         return runCount;
446     }
447
448     /**
449      * Returns the start index of the run at the given index, including attributes, styling, font, color, and background color.
450      *
451      * @param runIndex index of run
452      * @return int start index
453      */
454     int getRunStartWithAttributesAndStylingAndFontAndColorAndBackgroundColor(int runIndex) {
455         return runStarts[runIndex];
456     }
457
458     /**
459      * Returns the attribute keys of the run at the given index, including attributes, styling, font, color, and background color.
460      *
461      * @param runIndex index of run
462      * @return Vector<Attribute> attributes
463      */
464     Vector<Attribute> getRunAttributesWithAttributesAndStylingAndFontAndColorAndBackgroundColor(int runIndex) {
465         return runAttributes[runIndex];
466     }
467
468     /**
469      * Returns the attribute values of the run at the given index, including attributes, styling, font, color, and background color.
470      *
471      * @param runIndex index of run
472      * @return Vector<Object> values
473      */
474     Vector<Object> getRunAttributeValuesWithAttributesAndStylingAndFontAndColorAndBackgroundColor(int runIndex) {
475         return runAttributeValues[runIndex];
476     }
477
478     /**
479      * Returns the character at the given index, including attributes, styling, font, color, background color, and font style.
480      *
481      * @param index index of character
482      * @return String character
483      */
484     String getCharWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyle(int index) {
485         return text.charAt(index) + " ";
486     }
487
488     /**
489      * Returns the length of the text, including attributes, styling, font, color, background color, and font style.
490      *
491      * @return int length
492      */
493     int getLengthWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyle() {
494         return text.length() + runCount * 2;
495     }
496
497     /**
498      * Returns the number of runs in this AttributedString, including attributes, styling, font, color, background color, and font style.
499      *
500      * @return int runCount
501      */
502     int getRunCountWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyle() {
503         return runCount;
504     }
505
506     /**
507      * Returns the start index of the run at the given index, including attributes, styling, font, color, background color, and font style.
508      *
509      * @param runIndex index of run
510      * @return int start index
511      */
512     int getRunStartWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyle(int runIndex) {
513         return runStarts[runIndex];
514     }
515
516     /**
517      * Returns the attribute keys of the run at the given index, including attributes, styling, font, color, background color, and font style.
518      *
519      * @param runIndex index of run
520      * @return Vector<Attribute> attributes
521      */
522     Vector<Attribute> getRunAttributesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyle(int runIndex) {
523         return runAttributes[runIndex];
524     }
525
526     /**
527      * Returns the attribute values of the run at the given index, including attributes, styling, font, color, background color, and font style.
528      *
529      * @param runIndex index of run
530      * @return Vector<Object> values
531      */
532     Vector<Object> getRunAttributeValuesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyle(int runIndex) {
533         return runAttributeValues[runIndex];
534     }
535
536     /**
537      * Returns the character at the given index, including attributes, styling, font, color, background color, font style, and font weight.
538      *
539      * @param index index of character
540      * @return String character
541      */
542     String getCharWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeight(int index) {
543         return text.charAt(index) + " ";
544     }
545
546     /**
547      * Returns the length of the text, including attributes, styling, font, color, background color, font style, and font weight.
548      *
549      * @return int length
550      */
551     int getLengthWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeight() {
552         return text.length() + runCount * 2;
553     }
554
555     /**
556      * Returns the number of runs in this AttributedString, including attributes, styling, font, color, background color, font style, and font weight.
557      *
558      * @return int runCount
559      */
560     int getRunCountWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeight() {
561         return runCount;
562     }
563
564     /**
565      * Returns the start index of the run at the given index, including attributes, styling, font, color, background color, font style, and font weight.
566      *
567      * @param runIndex index of run
568      * @return int start index
569      */
570     int getRunStartWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeight(int runIndex) {
571         return runStarts[runIndex];
572     }
573
574     /**
575      * Returns the attribute keys of the run at the given index, including attributes, styling, font, color, background color, font style, and font weight.
576      *
577      * @param runIndex index of run
578      * @return Vector<Attribute> attributes
579      */
580     Vector<Attribute> getRunAttributesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeight(int runIndex) {
581         return runAttributes[runIndex];
582     }
583
584     /**
585      * Returns the attribute values of the run at the given index, including attributes, styling, font, color, background color, font style, and font weight.
586      *
587      * @param runIndex index of run
588      * @return Vector<Object> values
589      */
590     Vector<Object> getRunAttributeValuesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeight(int runIndex) {
591         return runAttributeValues[runIndex];
592     }
593
594     /**
595      * Returns the character at the given index, including attributes, styling, font, color, background color, font style, font weight, and font family.
596      *
597      * @param index index of character
598      * @return String character
599      */
600     String getCharWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamily(int index) {
601         return text.charAt(index) + " ";
602     }
603
604     /**
605      * Returns the length of the text, including attributes, styling, font, color, background color, font style, font weight, and font family.
606      *
607      * @return int length
608      */
609     int getLengthWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamily() {
610         return text.length() + runCount * 2;
611     }
612
613     /**
614      * Returns the number of runs in this AttributedString, including attributes, styling, font, color, background color, font style, font weight, and font family.
615      *
616      * @return int runCount
617      */
618     int getRunCountWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamily() {
619         return runCount;
620     }
621
622     /**
623      * Returns the start index of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, and font family.
624      *
625      * @param runIndex index of run
626      * @return int start index
627      */
628     int getRunStartWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamily(int runIndex) {
629         return runStarts[runIndex];
630     }
631
632     /**
633      * Returns the attribute keys of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, and font family.
634      *
635      * @param runIndex index of run
636      * @return Vector<Attribute> attributes
637      */
638     Vector<Attribute> getRunAttributesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamily(int runIndex) {
639         return runAttributes[runIndex];
640     }
641
642     /**
643      * Returns the attribute values of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, and font family.
644      *
645      * @param runIndex index of run
646      * @return Vector<Object> values
647      */
648     Vector<Object> getRunAttributeValuesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamily(int runIndex) {
649         return runAttributeValues[runIndex];
650     }
651
652     /**
653      * Returns the character at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, and font size.
654      *
655      * @param index index of character
656      * @return String character
657      */
658     String getCharWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSize(int index) {
659         return text.charAt(index) + " ";
660     }
661
662     /**
663      * Returns the length of the text, including attributes, styling, font, color, background color, font style, font weight, font family, and font size.
664      *
665      * @return int length
666      */
667     int getLengthWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSize() {
668         return text.length() + runCount * 2;
669     }
670
671     /**
672      * Returns the number of runs in this AttributedString, including attributes, styling, font, color, background color, font style, font weight, font family, and font size.
673      *
674      * @return int runCount
675      */
676     int getRunCountWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSize() {
677         return runCount;
678     }
679
680     /**
681      * Returns the start index of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, and font size.
682      *
683      * @param runIndex index of run
684      * @return int start index
685      */
686     int getRunStartWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSize(int runIndex) {
687         return runStarts[runIndex];
688     }
689
690     /**
691      * Returns the attribute keys of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, and font size.
692      *
693      * @param runIndex index of run
694      * @return Vector<Attribute> attributes
695      */
696     Vector<Attribute> getRunAttributesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSize(int runIndex) {
697         return runAttributes[runIndex];
698     }
699
700     /**
701      * Returns the attribute values of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, and font size.
702      *
703      * @param runIndex index of run
704      * @return Vector<Object> values
705      */
706     Vector<Object> getRunAttributeValuesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSize(int runIndex) {
707         return runAttributeValues[runIndex];
708     }
709
710     /**
711      * Returns the character at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, and font color.
712      *
713      * @param index index of character
714      * @return String character
715      */
716     String getCharWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColor(int index) {
717         return text.charAt(index) + " ";
718     }
719
720     /**
721      * Returns the length of the text, including attributes, styling, font, color, background color, font style, font weight, font family, font size, and font color.
722      *
723      * @return int length
724      */
725     int getLengthWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColor() {
726         return text.length() + runCount * 2;
727     }
728
729     /**
730      * Returns the number of runs in this AttributedString, including attributes, styling, font, color, background color, font style, font weight, font family, font size, and font color.
731      *
732      * @return int runCount
733      */
734     int getRunCountWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColor() {
735         return runCount;
736     }
737
738     /**
739      * Returns the start index of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, and font color.
740      *
741      * @param runIndex index of run
742      * @return int start index
743      */
744     int getRunStartWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColor(int runIndex) {
745         return runStarts[runIndex];
746     }
747
748     /**
749      * Returns the attribute keys of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, and font color.
750      *
751      * @param runIndex index of run
752      * @return Vector<Attribute> attributes
753      */
754     Vector<Attribute> getRunAttributesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColor(int runIndex) {
755         return runAttributes[runIndex];
756     }
757
758     /**
759      * Returns the attribute values of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, and font color.
760      *
761      * @param runIndex index of run
762      * @return Vector<Object> values
763      */
764     Vector<Object> getRunAttributeValuesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColor(int runIndex) {
765         return runAttributeValues[runIndex];
766     }
767
768     /**
769      * Returns the character at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, and font background color.
770      *
771      * @param index index of character
772      * @return String character
773      */
774     String getCharWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColor(int index) {
775         return text.charAt(index) + " ";
776     }
777
778     /**
779      * Returns the length of the text, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, and font background color.
780      *
781      * @return int length
782      */
783     int getLengthWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColor() {
784         return text.length() + runCount * 2;
785     }
786
787     /**
788      * Returns the number of runs in this AttributedString, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, and font background color.
789      *
790      * @return int runCount
791      */
792     int getRunCountWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColor() {
793         return runCount;
794     }
795
796     /**
797      * Returns the start index of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, and font background color.
798      *
799      * @param runIndex index of run
800      * @return int start index
801      */
802     int getRunStartWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColor(int runIndex) {
803         return runStarts[runIndex];
804     }
805
806     /**
807      * Returns the attribute keys of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, and font background color.
808      *
809      * @param runIndex index of run
810      * @return Vector<Attribute> attributes
811      */
812     Vector<Attribute> getRunAttributesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColor(int runIndex) {
813         return runAttributes[runIndex];
814     }
815
816     /**
817      * Returns the attribute values of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, and font background color.
818      *
819      * @param runIndex index of run
820      * @return Vector<Object> values
821      */
822     Vector<Object> getRunAttributeValuesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColor(int runIndex) {
823         return runAttributeValues[runIndex];
824     }
825
826     /**
827      * Returns the character at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, and font background style.
828      *
829      * @param index index of character
830      * @return String character
831      */
832     String getCharWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyle(int index) {
833         return text.charAt(index) + " ";
834     }
835
836     /**
837      * Returns the length of the text, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, and font background style.
838      *
839      * @return int length
840      */
841     int getLengthWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyle() {
842         return text.length() + runCount * 2;
843     }
844
845     /**
846      * Returns the number of runs in this AttributedString, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, and font background style.
847      *
848      * @return int runCount
849      */
850     int getRunCountWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyle() {
851         return runCount;
852     }
853
854     /**
855      * Returns the start index of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, and font background style.
856      *
857      * @param runIndex index of run
858      * @return int start index
859      */
860     int getRunStartWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyle(int runIndex) {
861         return runStarts[runIndex];
862     }
863
864     /**
865      * Returns the attribute keys of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, and font background style.
866      *
867      * @param runIndex index of run
868      * @return Vector<Attribute> attributes
869      */
870     Vector<Attribute> getRunAttributesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyle(int runIndex) {
871         return runAttributes[runIndex];
872     }
873
874     /**
875      * Returns the attribute values of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, and font background style.
876      *
877      * @param runIndex index of run
878      * @return Vector<Object> values
879      */
880     Vector<Object> getRunAttributeValuesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyle(int runIndex) {
881         return runAttributeValues[runIndex];
882     }
883
884     /**
885      * Returns the character at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, font background style, and font background font.
886      *
887      * @param index index of character
888      * @return String character
889      */
890     String getCharWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyleAndFontBackgroundFont(int index) {
891         return text.charAt(index) + " ";
892     }
893
894     /**
895      * Returns the length of the text, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, font background style, and font background font.
896      *
897      * @return int length
898      */
899     int getLengthWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyleAndFontBackgroundFont() {
900         return text.length() + runCount * 2;
901     }
902
903     /**
904      * Returns the number of runs in this AttributedString, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, font background style, and font background font.
905      *
906      * @return int runCount
907      */
908     int getRunCountWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyleAndFontBackgroundFont() {
909         return runCount;
910     }
911
912     /**
913      * Returns the start index of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, font background style, and font background font.
914      *
915      * @param runIndex index of run
916      * @return int start index
917      */
918     int getRunStartWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyleAndFontBackgroundFont(int runIndex) {
919         return runStarts[runIndex];
920     }
921
922     /**
923      * Returns the attribute keys of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, font background style, and font background font.
924      *
925      * @param runIndex index of run
926      * @return Vector<Attribute> attributes
927      */
928     Vector<Attribute> getRunAttributesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyleAndFontBackgroundFont(int runIndex) {
929         return runAttributes[runIndex];
930     }
931
932     /**
933      * Returns the attribute values of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, font background style, and font background font.
934      *
935      * @param runIndex index of run
936      * @return Vector<Object> values
937      */
938     Vector<Object> getRunAttributeValuesWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyleAndFontBackgroundFont(int runIndex) {
939         return runAttributeValues[runIndex];
940     }
941
942     /**
943      * Returns the character at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, font background style, font background font, and font background font color.
944      *
945      * @param index index of character
946      * @return String character
947      */
948     String getCharWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyleAndFontBackgroundFontAndFontBackgroundFontColor(int index) {
949         return text.charAt(index) + " ";
950     }
951
952     /**
953      * Returns the length of the text, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, font background style, font background font, and font background font color.
954      *
955      * @return int length
956      */
957     int getLengthWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyleAndFontBackgroundFontAndFontBackgroundFontColor() {
958         return text.length() + runCount * 2;
959     }
960
961     /**
962      * Returns the number of runs in this AttributedString, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, font background style, font background font, and font background font color.
963      *
964      * @return int runCount
965      */
966     int getRunCountWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyleAndFontBackgroundFontAndFontBackgroundFontColor() {
967         return runCount;
968     }
969
970     /**
971      * Returns the start index of the run at the given index, including attributes, styling, font, color, background color, font style, font weight, font family, font size, font color, font background color, font background style, font background font, and font background font color.
972      *
973      * @param runIndex index of run
974      * @return int start index
975      */
976     int getRunStartWithAttributesAndStylingAndFontAndColorAndBackgroundColorAndFontStyleAndFontWeightAndFontFamilyAndFontSizeAndFontColorAndFontBackgroundColorAndFontBackgroundStyleAndFontBackgroundFontAndFontBackgroundFontColor(int runIndex) {
977         return runStarts[runIndex];
978    
```

```

87     }
88
89     text = buffer.toString();
90
91     if (text.length() > 0) {
92         // Determine the runs, creating a new run when the attributes
93         // differ.
94         int offset = 0;
95         Map<Attribute, Object> last = null;
96
97         for (int counter = 0; counter < iterators.length; counter++) {
98             AttributedCharacterIterator iterator = iterators[counter];
99             int start = iterator.getBeginIndex();
100            int end = iterator.getEndIndex();
101            int index = start;
102
103            while (index < end) {
104                iterator.setIndex(index);
105
106                Map<Attribute, Object> attrs = iterator.getAttributes();
107
108                if (mapsDiffer(last, attrs)) {
109                    setAttributes(attrs, index - start + offset);
110                }
111                last = attrs;
112                index = iterator.getRunLimit();
113            }
114            offset += (end - start);
115        }
116    }
117 }
118 }
119
120 /**
121  * Constructs an AttributedString instance with the given text.
122  * @param text The text for this attributed string.
123  * @exception NullPointerException if <code>text</code> is null.
124  */
125 public AttributedString(String text) {
126     if (text == null) {
127         throw new NullPointerException();
128     }
129     this.text = text;
130 }
131
132 /**
133  * Constructs an AttributedString instance with the given text and attributes.
134  * @param text The text for this attributed string.
135  * @param attributes The attributes that apply to the entire string.
136  * @exception NullPointerException if <code>text</code> or
137  *         <code>attributes</code> is null.
138  * @exception IllegalArgumentException if the text has length 0
139  *         and the attributes parameter is not an empty Map (attributes

```

```

140     * cannot be applied to a 0-length range).
141     */
142     public AttributedString(String text,
143                             Map<? extends Attribute, ?> attributes)
144     {
145         if (text == null || attributes == null) {
146             throw new NullPointerException();
147         }
148         this.text = text;
149
150         if (text.length() == 0) {
151             if (attributes.isEmpty())
152                 return;
153             throw new IllegalArgumentException("Can't add attribute to 0-length text");
154         }
155
156         int attributeCount = attributes.size();
157         if (attributeCount > 0) {
158             createRunAttributeDataVectors();
159             Vector<Attribute> newRunAttributes = new Vector<>(attributeCount);
160             Vector<Object> newRunAttributeValues = new Vector<>(attributeCount);
161             runAttributes[0] = newRunAttributes;
162             runAttributeValues[0] = newRunAttributeValues;
163
164             Iterator<? extends Map.Entry<? extends Attribute, ?>> iterator = attributes.entrySet().
165                 iterator();
166             while (iterator.hasNext()) {
167                 Map.Entry<? extends Attribute, ?> entry = iterator.next();
168                 newRunAttributes.addElement(entry.getKey());
169                 newRunAttributeValues.addElement(entry.getValue());
170             }
171         }
172     }
173
174     /**
175     * Constructs an AttributedString instance with the given attributed
176     * text represented by AttributedCharacterIterator.
177     * @param text The text for this attributed string.
178     * @exception NullPointerException if <code>text</code> is null.
179     */
180     public AttributedString(AttributedCharacterIterator text) {
181         // If performance is critical, this constructor should be
182         // implemented here rather than invoking the constructor for a
183         // subrange. We can avoid some range checking in the loops.
184         this(text, text.getBeginIndex(), text.getEndIndex(), null);
185     }
186
187     /**
188     * Constructs an AttributedString instance with the subrange of
189     * the given attributed text represented by
190     * AttributedCharacterIterator. If the given range produces an
191     * empty text, all attributes will be discarded. Note that any
192     * attributes wrapped by an Annotation object are discarded for a

```

```

192     * subrange of the original attribute range.
193     *
194     * @param text The text for this attributed string.
195     * @param beginIndex Index of the first character of the range.
196     * @param endIndex Index of the character following the last character
197     * of the range.
198     * @exception NullPointerException if <code>text</code> is null.
199     * @exception IllegalArgumentException if the subrange given by
200     * beginIndex and endIndex is out of the text range.
201     * @see java.text.Annotation
202     */
203     public AttributedString(AttributedCharacterIterator text,
204                             int beginIndex,
205                             int endIndex) {
206         this(text, beginIndex, endIndex, null);
207     }
208
209     /**
210     * Constructs an AttributedString instance with the subrange of
211     * the given attributed text represented by
212     * AttributedCharacterIterator. Only attributes that match the
213     * given attributes will be incorporated into the instance. If the
214     * given range produces an empty text, all attributes will be
215     * discarded. Note that any attributes wrapped by an Annotation
216     * object are discarded for a subrange of the original attribute
217     * range.
218     *
219     * @param text The text for this attributed string.
220     * @param beginIndex Index of the first character of the range.
221     * @param endIndex Index of the character following the last character
222     * of the range.
223     * @param attributes Specifies attributes to be extracted
224     * from the text. If null is specified, all available attributes will
225     * be used.
226     * @exception NullPointerException if <code>text</code> is null.
227     * @exception IllegalArgumentException if the subrange given by
228     * beginIndex and endIndex is out of the text range.
229     * @see java.text.Annotation
230     */
231     public AttributedString(AttributedCharacterIterator text,
232                             int beginIndex,
233                             int endIndex,
234                             Attribute[] attributes) {
235         if (text == null) {
236             throw new NullPointerException();
237         }
238
239         // Validate the given subrange
240         int textBeginIndex = text.getBeginIndex();
241         int textEndIndex = text.getEndIndex();
242         if (beginIndex < textBeginIndex || endIndex > textEndIndex || beginIndex > endIndex)
243             throw new IllegalArgumentException("Invalid substring range");
244

```

```

245     // Copy the given string
246     StringBuffer textBuffer = new StringBuffer();
247     text.setIndex(beginIndex);
248     for (char c = text.current(); text.getIndex() < endIndex; c = text.next())
249         textBuffer.append(c);
250     this.text = textBuffer.toString();
251
252     if (beginIndex == endIndex)
253         return;
254
255     // Select attribute keys to be taken care of
256     HashSet<Attribute> keys = new HashSet<>();
257     if (attributes == null) {
258         keys.addAll(text.getAllAttributeKeys());
259     } else {
260         for (int i = 0; i < attributes.length; i++)
261             keys.add(attributes[i]);
262         keys.retainAll(text.getAllAttributeKeys());
263     }
264     if (keys.isEmpty())
265         return;
266
267     // Get and set attribute runs for each attribute name. Need to
268     // scan from the top of the text so that we can discard any
269     // Annotation that is no longer applied to a subset text segment.
270     Iterator<Attribute> itr = keys.iterator();
271     while (itr.hasNext()) {
272         Attribute attributeKey = itr.next();
273         text.setIndex(textBeginIndex);
274         while (text.getIndex() < endIndex) {
275             int start = text.getRunStart(attributeKey);
276             int limit = text.getRunLimit(attributeKey);
277             Object value = text.getAttribute(attributeKey);
278
279             if (value != null) {
280                 if (value instanceof Annotation) {
281                     if (start >= beginIndex && limit <= endIndex) {
282                         addAttribute(attributeKey, value, start - beginIndex, limit -
283                             beginIndex);
284                     } else {
285                         if (limit > endIndex)
286                             break;
287                     }
288                 } else {
289                     // if the run is beyond the given (subset) range, we
290                     // don't need to process further.
291                     if (start >= endIndex)
292                         break;
293                     if (limit > beginIndex) {
294                         // attribute is applied to any subrange
295                         if (start < beginIndex)
296                             start = beginIndex;
297                         if (limit > endIndex)

```



```

297         limit = endIndex;
298         if (start != limit) {
299             addAttribute(attributeKey, value, start - beginIndex, limit -
300                             beginIndex);
301         }
302     }
303 }
304 text.setIndex(limit);
305 }
306 }
307 }
308
309 /**
310  * Adds an attribute to the entire string.
311  * @param attribute the attribute key
312  * @param value the value of the attribute; may be null
313  * @exception NullPointerException if <code>attribute</code> is null.
314  * @exception IllegalArgumentException if the AttributedString has length 0
315  * (attributes cannot be applied to a 0-length range).
316  */
317 public void addAttribute(Attribute attribute, Object value) {
318
319     if (attribute == null) {
320         throw new NullPointerException();
321     }
322
323     int len = length();
324     if (len == 0) {
325         throw new IllegalArgumentException("Can't add attribute to 0-length text");
326     }
327
328     addAttributeImpl(attribute, value, 0, len);
329 }
330
331 /**
332  * Adds an attribute to a subrange of the string.
333  * @param attribute the attribute key
334  * @param value The value of the attribute. May be null.
335  * @param beginIndex Index of the first character of the range.
336  * @param endIndex Index of the character following the last character of the range.
337  * @exception NullPointerException if <code>attribute</code> is null.
338  * @exception IllegalArgumentException if beginIndex is less than 0, endIndex is
339  * greater than the length of the string, or beginIndex and endIndex together don't
340  * define a non-empty subrange of the string.
341  */
342 public void addAttribute(Attribute attribute, Object value,
343                         int beginIndex, int endIndex) {
344
345     if (attribute == null) {
346         throw new NullPointerException();
347     }
348

```

```

349     if (beginIndex < 0 || endIndex > length() || beginIndex >= endIndex) {
350         throw new IllegalArgumentException("Invalid substring range");
351     }
352
353     addAttributeImpl(attribute, value, beginIndex, endIndex);
354 }
355
356 /**
357  * Adds a set of attributes to a subrange of the string.
358  * @param attributes The attributes to be added to the string.
359  * @param beginIndex Index of the first character of the range.
360  * @param endIndex Index of the character following the last
361  * character of the range.
362  * @exception NullPointerException if <code>attributes</code> is null.
363  * @exception IllegalArgumentException if beginIndex is less than
364  * 0, endIndex is greater than the length of the string, or
365  * beginIndex and endIndex together don't define a non-empty
366  * subrange of the string and the attributes parameter is not an
367  * empty Map.
368  */
369 public void addAttributes(Map<? extends Attribute, ?> attributes,
370                          int beginIndex, int endIndex)
371 {
372     if (attributes == null) {
373         throw new NullPointerException();
374     }
375
376     if (beginIndex < 0 || endIndex > length() || beginIndex > endIndex) {
377         throw new IllegalArgumentException("Invalid substring range");
378     }
379     if (beginIndex == endIndex) {
380         if (attributes.isEmpty())
381             return;
382         throw new IllegalArgumentException("Can't add attribute to 0-length text");
383     }
384
385     // make sure we have run attribute data vectors
386     if (runCount == 0) {
387         createRunAttributeDataVectors();
388     }
389
390     // break up runs if necessary
391     int beginRunIndex = ensureRunBreak(beginIndex);
392     int endRunIndex = ensureRunBreak(endIndex);
393
394     Iterator<? extends Map.Entry<? extends Attribute, ?>> iterator =
395         attributes.entrySet().iterator();
396     while (iterator.hasNext()) {
397         Map.Entry<? extends Attribute, ?> entry = iterator.next();
398         addAttributeRunData(entry.getKey(), entry.getValue(), beginRunIndex, endRunIndex);
399     }
400 }
401

```

```

402 private synchronized void addAttributeImpl(Attribute attribute, Object value,
403     int beginIndex, int endIndex) {
404
405     // make sure we have run attribute data vectors
406     if (runCount == 0) {
407         createRunAttributeDataVectors();
408     }
409
410     // break up runs if necessary
411     int beginRunIndex = ensureRunBreak(beginIndex);
412     int endRunIndex = ensureRunBreak(endIndex);
413
414     addAttributeRunData(attribute, value, beginRunIndex, endRunIndex);
415 }
416
417 private final void createRunAttributeDataVectors() {
418     // use temporary variables so things remain consistent in case of an exception
419     int newRunStarts[] = new int[ARRAY_SIZE_INCREMENT];
420
421     @SuppressWarnings("unchecked")
422     Vector<Attribute> newRunAttributes[] = (Vector<Attribute>[]) new Vector<?>[
423         ARRAY_SIZE_INCREMENT];
424
425     @SuppressWarnings("unchecked")
426     Vector<Object> newRunAttributeValues[] = (Vector<Object>[]) new Vector<?>[
427         ARRAY_SIZE_INCREMENT];
428
429     runStarts = newRunStarts;
430     runAttributes = newRunAttributes;
431     runAttributeValues = newRunAttributeValues;
432     runArraySize = ARRAY_SIZE_INCREMENT;
433     runCount = 1; // assume initial run starting at index 0
434 }
435
436 // ensure there's a run break at offset, return the index of the run
437 private final int ensureRunBreak(int offset) {
438     return ensureRunBreak(offset, true);
439 }
440
441 /**
442  * Ensures there is a run break at offset, returning the index of
443  * the run. If this results in splitting a run, two things can happen:
444  * <ul>
445  * <li>If copyAttrs is true, the attributes from the existing run
446  *     will be placed in both of the newly created runs.
447  * <li>If copyAttrs is false, the attributes from the existing run
448  *     will NOT be copied to the run to the right (>= offset) of the break,
449  *     but will exist on the run to the left (< offset).
450  * </ul>
451  */
452 private final int ensureRunBreak(int offset, boolean copyAttrs) {
453     if (offset == length()) {
454         return runCount;
455     }

```

```

453     }
454
455     // search for the run index where this offset should be
456     int runIndex = 0;
457     while (runIndex < runCount && runStarts[runIndex] < offset) {
458         runIndex++;
459     }
460
461     // if the offset is at a run start already, we're done
462     if (runIndex < runCount && runStarts[runIndex] == offset) {
463         return runIndex;
464     }
465
466     // we'll have to break up a run
467     // first, make sure we have enough space in our arrays
468     if (runCount == runArraySize) {
469         int newArraySize = runArraySize + ARRAY_SIZE_INCREMENT;
470         int newRunStarts[] = new int[newArraySize];
471
472         @SuppressWarnings("unchecked")
473         Vector<Attribute> newRunAttributes[] = (Vector<Attribute>[]) new Vector<?>[newArraySize
474             ];
475
476         @SuppressWarnings("unchecked")
477         Vector<Object> newRunAttributeValues[] = (Vector<Object>[]) new Vector<?>[newArraySize
478             ];
479
480         for (int i = 0; i < runArraySize; i++) {
481             newRunStarts[i] = runStarts[i];
482             newRunAttributes[i] = runAttributes[i];
483             newRunAttributeValues[i] = runAttributeValues[i];
484         }
485         runStarts = newRunStarts;
486         runAttributes = newRunAttributes;
487         runAttributeValues = newRunAttributeValues;
488         runArraySize = newArraySize;
489     }
490
491     // make copies of the attribute information of the old run that the new one used to be part
492     // of
493     // use temporary variables so things remain consistent in case of an exception
494     Vector<Attribute> newRunAttributes = null;
495     Vector<Object> newRunAttributeValues = null;
496
497     if (copyAttrs) {
498         Vector<Attribute> oldRunAttributes = runAttributes[runIndex - 1];
499         Vector<Object> oldRunAttributeValues = runAttributeValues[runIndex - 1];
500         if (oldRunAttributes != null) {
501             newRunAttributes = new Vector<>(oldRunAttributes);
502         }
503         if (oldRunAttributeValues != null) {
504             newRunAttributeValues = new Vector<>(oldRunAttributeValues);
505         }
506     }

```

```

503     }
504
505     // now actually break up the run
506     runCount++;
507     for (int i = runCount - 1; i > runIndex; i--) {
508         runStarts[i] = runStarts[i - 1];
509         runAttributes[i] = runAttributes[i - 1];
510         runAttributeValues[i] = runAttributeValues[i - 1];
511     }
512     runStarts[runIndex] = offset;
513     runAttributes[runIndex] = newRunAttributes;
514     runAttributeValues[runIndex] = newRunAttributeValues;
515
516     return runIndex;
517 }
518
519 // add the attribute attribute/value to all runs where beginRunIndex <= runIndex < endRunIndex
520 private void addAttributeRunData(Attribute attribute, Object value,
521     int beginRunIndex, int endRunIndex) {
522
523     for (int i = beginRunIndex; i < endRunIndex; i++) {
524         int keyValueIndex = -1; // index of key and value in our vectors; assume we don't have
525                                 // an entry yet
526         if (runAttributes[i] == null) {
527             Vector<Attribute> newRunAttributes = new Vector<>();
528             Vector<Object> newRunAttributeValues = new Vector<>();
529             runAttributes[i] = newRunAttributes;
530             runAttributeValues[i] = newRunAttributeValues;
531         } else {
532             // check whether we have an entry already
533             keyValueIndex = runAttributes[i].indexOf(attribute);
534         }
535
536         if (keyValueIndex == -1) {
537             // create new entry
538             int oldSize = runAttributes[i].size();
539             runAttributes[i].addElement(attribute);
540             try {
541                 runAttributeValues[i].addElement(value);
542             } catch (Exception e) {
543                 runAttributes[i].setSize(oldSize);
544                 runAttributeValues[i].setSize(oldSize);
545             }
546         } else {
547             // update existing entry
548             runAttributeValues[i].set(keyValueIndex, value);
549         }
550     }
551 }
552
553 /**
554  * Creates an AttributedStringIterator instance that provides access to the entire contents

```

```

        of
555     * this string.
556     *
557     * @return An iterator providing access to the text and its attributes.
558     */
559     public AttributedStringIterator getIterator() {
560         return getIterator(null, 0, length());
561     }
562
563     /**
564     * Creates an AttributedStringIterator instance that provides access to
565     * selected contents of this string.
566     * Information about attributes not listed in attributes that the
567     * implementor may have need not be made accessible through the iterator.
568     * If the list is null, all available attribute information should be made
569     * accessible.
570     *
571     * @param attributes a list of attributes that the client is interested in
572     * @return an iterator providing access to the entire text and its selected attributes
573     */
574     public AttributedStringIterator getIterator(Attribute[] attributes) {
575         return getIterator(attributes, 0, length());
576     }
577
578     /**
579     * Creates an AttributedStringIterator instance that provides access to
580     * selected contents of this string.
581     * Information about attributes not listed in attributes that the
582     * implementor may have need not be made accessible through the iterator.
583     * If the list is null, all available attribute information should be made
584     * accessible.
585     *
586     * @param attributes a list of attributes that the client is interested in
587     * @param beginIndex the index of the first character
588     * @param endIndex the index of the character following the last character
589     * @return an iterator providing access to the text and its attributes
590     * @exception IllegalArgumentException if beginIndex is less than 0,
591     *         endIndex is greater than the length of the string, or beginIndex is
592     *         greater than endIndex.
593     */
594     public AttributedStringIterator getIterator(Attribute[] attributes, int beginIndex, int
        endIndex) {
595         return new AttributedStringIterator(attributes, beginIndex, endIndex);
596     }
597
598     // all (with the exception of length) reading operations are private,
599     // since AttributedString instances are accessed through iterators.
600
601     // length is package private so that CharacterIteratorFieldDelegate can
602     // access it without creating an AttributedStringIterator.
603     int length() {
604         return text.length();
605     }

```

```

606
607 private char charAt(int index) {
608     return text.charAt(index);
609 }
610
611 private synchronized Object getAttribute(Attribute attribute, int runIndex) {
612     Vector<Attribute> currentRunAttributes = runAttributes[runIndex];
613     Vector<Object> currentRunAttributeValues = runAttributeValues[runIndex];
614     if (currentRunAttributes == null) {
615         return null;
616     }
617     int attributeIndex = currentRunAttributes.indexOf(attribute);
618     if (attributeIndex != -1) {
619         return currentRunAttributeValues.elementAt(attributeIndex);
620     }
621     else {
622         return null;
623     }
624 }
625
626 // gets an attribute value, but returns an annotation only if it's range does not extend
627 // outside the range beginIndex..endIndex
628 private Object getAttributeCheckRange(Attribute attribute, int runIndex, int beginIndex, int
629     endIndex) {
630     Object value = getAttribute(attribute, runIndex);
631     if (value instanceof Annotation) {
632         // need to check whether the annotation's range extends outside the iterator's range
633         if (beginIndex > 0) {
634             int currIndex = runIndex;
635             int runStart = runStarts[currIndex];
636             while (runStart >= beginIndex &&
637                 valuesMatch(value, getAttribute(attribute, currIndex - 1))) {
638                 currIndex--;
639                 runStart = runStarts[currIndex];
640             }
641             if (runStart < beginIndex) {
642                 // annotation's range starts before iterator's range
643                 return null;
644             }
645         }
646         int textLength = length();
647         if (endIndex < textLength) {
648             int currIndex = runIndex;
649             int runLimit = (currIndex < runCount - 1) ? runStarts[currIndex + 1] : textLength;
650             while (runLimit <= endIndex &&
651                 valuesMatch(value, getAttribute(attribute, currIndex + 1))) {
652                 currIndex++;
653                 runLimit = (currIndex < runCount - 1) ? runStarts[currIndex + 1] : textLength;
654             }
655             if (runLimit > endIndex) {
656                 // annotation's range ends after iterator's range
657                 return null;
658             }
659         }
660     }

```

```

657     }
658     // annotation's range is subrange of iterator's range,
659     // so we can return the value
660 }
661 return value;
662 }
663
664 // returns whether all specified attributes have equal values in the runs with the given
665 // indices
666 private boolean attributeValuesMatch(Set<? extends Attribute> attributes, int runIndex1, int
667 runIndex2) {
668     Iterator<? extends Attribute> iterator = attributes.iterator();
669     while (iterator.hasNext()) {
670         Attribute key = iterator.next();
671         if (!valuesMatch(getAttribute(key, runIndex1), getAttribute(key, runIndex2))) {
672             return false;
673         }
674     }
675     return true;
676 }
677
678 // returns whether the two objects are either both null or equal
679 private final static boolean valuesMatch(Object value1, Object value2) {
680     if (value1 == null) {
681         return value2 == null;
682     } else {
683         return value1.equals(value2);
684     }
685 }
686
687 /**
688  * Appends the contents of the CharacterIterator iterator into the
689  * StringBuffer buf.
690  */
691 private final void appendContents(StringBuffer buf,
692 CharacterIterator iterator) {
693     int index = iterator.getBeginIndex();
694     int end = iterator.getEndIndex();
695
696     while (index < end) {
697         iterator.setIndex(index++);
698         buf.append(iterator.current());
699     }
700 }
701
702 /**
703  * Sets the attributes for the range from offset to the next run break
704  * (typically the end of the text) to the ones specified in attrs.
705  * This is only meant to be called from the constructor!
706  */
707 private void setAttributes(Map<Attribute, Object> attrs, int offset) {
708     if (runCount == 0) {
709         createRunAttributeDataVectors();

```



```

708     }
709
710     int index = ensureRunBreak(offset, false);
711     int size;
712
713     if (attrs != null && (size = attrs.size()) > 0) {
714         Vector<Attribute> runAttrs = new Vector<>(size);
715         Vector<Object> runValues = new Vector<>(size);
716         Iterator<Map.Entry<Attribute, Object>> iterator = attrs.entrySet().iterator();
717
718         while (iterator.hasNext()) {
719             Map.Entry<Attribute, Object> entry = iterator.next();
720
721             runAttrs.add(entry.getKey());
722             runValues.add(entry.getValue());
723         }
724         runAttributes[index] = runAttrs;
725         runAttributeValues[index] = runValues;
726     }
727 }
728
729 /**
730  * Returns true if the attributes specified in last and attrs differ.
731  */
732 private static <K,V> boolean mapsDiffer(Map<K, V> last, Map<K, V> attrs) {
733     if (last == null) {
734         return (attrs != null && attrs.size() > 0);
735     }
736     return (!last.equals(attrs));
737 }
738
739
740 // the iterator class associated with this string class
741
742 final private class AttributedStringIterator implements AttributedCharacterIterator {
743
744     // note on synchronization:
745     // we don't synchronize on the iterator, assuming that an iterator is only used in one
746     // thread.
747     // we do synchronize access to the AttributedString however, since it's more likely to be
748     // shared between threads.
749
750     // start and end index for our iteration
751     private int beginIndex;
752     private int endIndex;
753
754     // attributes that our client is interested in
755     private Attribute[] relevantAttributes;
756
757     // the current index for our iteration
758     // invariant: beginIndex <= currentIndex <= endIndex
759     private int currentIndex;

```

```
759 // information about the run that includes currentIndex
760 private int currentRunIndex;
761 private int currentRunStart;
762 private int currentRunLimit;
763
764 // constructor
765 AttributedStringIterator(Attribute[] attributes, int beginIndex, int endIndex) {
766
767     if (beginIndex < 0 || beginIndex > endIndex || endIndex > length()) {
768         throw new IllegalArgumentException("Invalid substring range");
769     }
770
771     this.beginIndex = beginIndex;
772     this.endIndex = endIndex;
773     this.currentIndex = beginIndex;
774     updateRunInfo();
775     if (attributes != null) {
776         relevantAttributes = attributes.clone();
777     }
778 }
779
780 // Object methods. See documentation in that class.
781
782 public boolean equals(Object obj) {
783     if (this == obj) {
784         return true;
785     }
786     if (!(obj instanceof AttributedStringIterator)) {
787         return false;
788     }
789
790     AttributedStringIterator that = (AttributedStringIterator) obj;
791
792     if (AttributedString.this != that.getString())
793         return false;
794     if (currentIndex != that.currentIndex || beginIndex != that.beginIndex || endIndex !=
795         that.endIndex)
796         return false;
797     return true;
798 }
799
800 public int hashCode() {
801     return text.hashCode() ^ currentIndex ^ beginIndex ^ endIndex;
802 }
803
804 public Object clone() {
805     try {
806         AttributedStringIterator other = (AttributedStringIterator) super.clone();
807         return other;
808     } catch (CloneNotSupportedException e) {
809         throw new InternalError(e);
810     }
811 }
```

```
811     }
812
813     // CharacterIterator methods. See documentation in that interface.
814
815     public char first() {
816         return internalSetIndex(beginIndex);
817     }
818
819     public char last() {
820         if (endIndex == beginIndex) {
821             return internalSetIndex(endIndex);
822         } else {
823             return internalSetIndex(endIndex - 1);
824         }
825     }
826
827     public char current() {
828         if (currentIndex == endIndex) {
829             return DONE;
830         } else {
831             return charAt(currentIndex);
832         }
833     }
834
835     public char next() {
836         if (currentIndex < endIndex) {
837             return internalSetIndex(currentIndex + 1);
838         }
839         else {
840             return DONE;
841         }
842     }
843
844     public char previous() {
845         if (currentIndex > beginIndex) {
846             return internalSetIndex(currentIndex - 1);
847         }
848         else {
849             return DONE;
850         }
851     }
852
853     public char setIndex(int position) {
854         if (position < beginIndex || position > endIndex)
855             throw new IllegalArgumentException("Invalid index");
856         return internalSetIndex(position);
857     }
858
859     public int getBeginIndex() {
860         return beginIndex;
861     }
862
863     public int getEndIndex() {
```

```
864         return endIndex;
865     }
866
867     public int getIndex() {
868         return currentIndex;
869     }
870
871     // AttributedCharacterIterator methods. See documentation in that interface.
872
873     public int getRunStart() {
874         return currentRunStart;
875     }
876
877     public int getRunStart(Attribute attribute) {
878         if (currentRunStart == beginIndex || currentRunIndex == -1) {
879             return currentRunStart;
880         } else {
881             Object value = getAttribute(attribute);
882             int runStart = currentRunStart;
883             int runIndex = currentRunIndex;
884             while (runStart > beginIndex &&
885                 valuesMatch(value, AttributedString.this.getAttribute(attribute, runIndex -
886                     1))) {
887                 runIndex--;
888                 runStart = runStarts[runIndex];
889             }
890             if (runStart < beginIndex) {
891                 runStart = beginIndex;
892             }
893             return runStart;
894         }
895     }
896
897     public int getRunStart(Set<? extends Attribute> attributes) {
898         if (currentRunStart == beginIndex || currentRunIndex == -1) {
899             return currentRunStart;
900         } else {
901             int runStart = currentRunStart;
902             int runIndex = currentRunIndex;
903             while (runStart > beginIndex &&
904                 AttributedString.this.attributeValuesMatch(attributes, currentRunIndex,
905                     runIndex - 1)) {
906                 runIndex--;
907                 runStart = runStarts[runIndex];
908             }
909             if (runStart < beginIndex) {
910                 runStart = beginIndex;
911             }
912             return runStart;
913         }
914     }
915
916     public int getRunLimit() {
```

```

915         return currentRunLimit;
916     }
917
918     public int getRunLimit(Attribute attribute) {
919         if (currentRunLimit == endIndex || currentRunIndex == -1) {
920             return currentRunLimit;
921         } else {
922             Object value = getAttribute(attribute);
923             int runLimit = currentRunLimit;
924             int runIndex = currentRunIndex;
925             while (runLimit < endIndex &&
926                 valuesMatch(value, AttributedString.this.getAttribute(attribute, runIndex +
927                     1))) {
928                 runIndex++;
929                 runLimit = runIndex < runCount - 1 ? runStarts[runIndex + 1] : endIndex;
930             }
931             if (runLimit > endIndex) {
932                 runLimit = endIndex;
933             }
934             return runLimit;
935         }
936     }
937
938     public int getRunLimit(Set<? extends Attribute> attributes) {
939         if (currentRunLimit == endIndex || currentRunIndex == -1) {
940             return currentRunLimit;
941         } else {
942             int runLimit = currentRunLimit;
943             int runIndex = currentRunIndex;
944             while (runLimit < endIndex &&
945                 AttributedString.this.attributeValuesMatch(attributes, currentRunIndex,
946                     runIndex + 1)) {
947                 runIndex++;
948                 runLimit = runIndex < runCount - 1 ? runStarts[runIndex + 1] : endIndex;
949             }
950             if (runLimit > endIndex) {
951                 runLimit = endIndex;
952             }
953             return runLimit;
954         }
955     }
956
957     public Map<Attribute, Object> getAttributes() {
958         if (runAttributes == null || currentRunIndex == -1 || runAttributes[currentRunIndex] ==
959             null) {
960             // ??? would be nice to return null, but current spec doesn't allow it
961             // returning Hashtable saves AttributeMap from dealing with emptiness
962             return new Hashtable<>();
963         }
964         return new AttributeMap(currentRunIndex, beginIndex, endIndex);
965     }
966
967     public Set<Attribute> getAllAttributeKeys() {

```

```

965     // ??? This should screen out attribute keys that aren't relevant to the client
966     if (runAttributes == null) {
967         // ??? would be nice to return null, but current spec doesn't allow it
968         // returning HashSet saves us from dealing with emptiness
969         return new HashSet<>();
970     }
971     synchronized (AttributedString.this) {
972         // ??? should try to create this only once, then update if necessary,
973         // and give callers read-only view
974         Set<Attribute> keys = new HashSet<>();
975         int i = 0;
976         while (i < runCount) {
977             if (runStarts[i] < endIndex && (i == runCount - 1 || runStarts[i + 1] >
978                 beginIndex)) {
979                 Vector<Attribute> currentRunAttributes = runAttributes[i];
980                 if (currentRunAttributes != null) {
981                     int j = currentRunAttributes.size();
982                     while (j-- > 0) {
983                         keys.add(currentRunAttributes.get(j));
984                     }
985                 }
986                 i++;
987             }
988             return keys;
989         }
990     }
991
992     public Object getAttribute(Attribute attribute) {
993         int runIndex = currentRunIndex;
994         if (runIndex < 0) {
995             return null;
996         }
997         return AttributedString.this.getAttributeCheckRange(attribute, runIndex, beginIndex,
998             endIndex);
999     }
1000
1001     // internally used methods
1002
1003     private AttributedString getString() {
1004         return AttributedString.this;
1005     }
1006
1007     // set the current index, update information about the current run if necessary,
1008     // return the character at the current index
1009     private char internalSetIndex(int position) {
1010         currentIndex = position;
1011         if (position < currentRunStart || position >= currentRunLimit) {
1012             updateRunInfo();
1013         }
1014         if (currentIndex == endIndex) {
1015             return DONE;
1016         } else {

```

```

1016         return charAt(position);
1017     }
1018 }
1019
1020 // update the information about the current run
1021 private void updateRunInfo() {
1022     if (currentIndex == endIndex) {
1023         currentRunStart = currentRunLimit = endIndex;
1024         currentRunIndex = -1;
1025     } else {
1026         synchronized (AttributedString.this) {
1027             int runIndex = -1;
1028             while (runIndex < runCount - 1 && runStarts[runIndex + 1] <= currentIndex)
1029                 runIndex++;
1030             currentRunIndex = runIndex;
1031             if (runIndex >= 0) {
1032                 currentRunStart = runStarts[runIndex];
1033                 if (currentRunStart < beginIndex)
1034                     currentRunStart = beginIndex;
1035             }
1036             else {
1037                 currentRunStart = beginIndex;
1038             }
1039             if (runIndex < runCount - 1) {
1040                 currentRunLimit = runStarts[runIndex + 1];
1041                 if (currentRunLimit > endIndex)
1042                     currentRunLimit = endIndex;
1043             }
1044             else {
1045                 currentRunLimit = endIndex;
1046             }
1047         }
1048     }
1049 }
1050
1051 }
1052
1053 // the map class associated with this string class, giving access to the attributes of one run
1054
1055 final private class AttributeMap extends AbstractMap<Attribute, Object> {
1056
1057     int runIndex;
1058     int beginIndex;
1059     int endIndex;
1060
1061     AttributeMap(int runIndex, int beginIndex, int endIndex) {
1062         this.runIndex = runIndex;
1063         this.beginIndex = beginIndex;
1064         this.endIndex = endIndex;
1065     }
1066
1067     public Set<Map.Entry<Attribute, Object>> entrySet() {
1068         HashSet<Map.Entry<Attribute, Object>> set = new HashSet<>();

```

```

1069     synchronized (AttributedString.this) {
1070         int size = runAttributes[runIndex].size();
1071         for (int i = 0; i < size; i++) {
1072             Attribute key = runAttributes[runIndex].get(i);
1073             Object value = runAttributeValues[runIndex].get(i);
1074             if (value instanceof Annotation) {
1075                 value = AttributedString.this.getAttributeCheckRange(key,
1076                                     runIndex, beginIndex, endIndex);
1077                 if (value == null) {
1078                     continue;
1079                 }
1080             }
1081             Map.Entry<Attribute, Object> entry = new AttributeEntry(key, value);
1082             set.add(entry);
1083         }
1084     }
1085     return set;
1086 }
1087
1088 public Object get(Object key) {
1089     return AttributedString.this.getAttributeCheckRange((Attribute) key, runIndex,
1090     beginIndex, endIndex);
1091 }
1092 }
1093 }
1094
1095 class AttributeEntry implements Map.Entry<Attribute, Object> {
1096
1097     private Attribute key;
1098     private Object value;
1099
1100     AttributeEntry(Attribute key, Object value) {
1101         this.key = key;
1102         this.value = value;
1103     }
1104
1105     public boolean equals(Object o) {
1106         if (!(o instanceof AttributeEntry)) {
1107             return false;
1108         }
1109         AttributeEntry other = (AttributeEntry) o;
1110         return other.key.equals(key) &&
1111             (value == null ? other.value == null : other.value.equals(value));
1112     }
1113
1114     public Attribute getKey() {
1115         return key;
1116     }
1117
1118     public Object getValue() {
1119         return value;
1120     }

```



```

1121
1122     public Object setValue(Object newValue) {
1123         throw new UnsupportedOperationException();
1124     }
1125
1126     public int hashCode() {
1127         return key.hashCode() ^ (value==null ? 0 : value.hashCode());
1128     }
1129
1130     public String toString() {
1131         return key.toString()+"="+value.toString();
1132     }
1133 }

```

2.4 Bidi.java

Secuencia 13: java/text/Bidi.java

```

1  /*
2  * Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright IBM Corp. 1999-2003 - All Rights Reserved
28 *
29 * The original version of this source code and documentation is
30 * copyrighted and owned by IBM. These materials are provided
31 * under terms of a License Agreement between IBM and Sun.
32 * This technology is protected by multiple US and International
33 * patents. This notice and attribution to IBM may not be removed.
34 */
35
36 package java.text;

```

```

37
38 import sun.text.bidi.BidiBase;
39
40 /**
41  * This class implements the Unicode Bidirectional Algorithm.
42  * <p>
43  * A Bidi object provides information on the bidirectional reordering of the text
44  * used to create it. This is required, for example, to properly display Arabic
45  * or Hebrew text. These languages are inherently mixed directional, as they order
46  * numbers from left-to-right while ordering most other text from right-to-left.
47  * <p>
48  * Once created, a Bidi object can be queried to see if the text it represents is
49  * all left-to-right or all right-to-left. Such objects are very lightweight and
50  * this text is relatively easy to process.
51  * <p>
52  * If there are multiple runs of text, information about the runs can be accessed
53  * by indexing to get the start, limit, and level of a run. The level represents
54  * both the direction and the 'nesting level' of a directional run. Odd levels
55  * are right-to-left, while even levels are left-to-right. So for example level
56  * 0 represents left-to-right text, while level 1 represents right-to-left text, and
57  * level 2 represents left-to-right text embedded in a right-to-left run.
58  *
59  * @since 1.4
60  */
61 public final class Bidi {
62
63     /** Constant indicating base direction is left-to-right. */
64     public static final int DIRECTION_LEFT_TO_RIGHT = 0;
65
66     /** Constant indicating base direction is right-to-left. */
67     public static final int DIRECTION_RIGHT_TO_LEFT = 1;
68
69     /**
70      * Constant indicating that the base direction depends on the first strong
71      * directional character in the text according to the Unicode
72      * Bidirectional Algorithm. If no strong directional character is present,
73      * the base direction is left-to-right.
74      */
75     public static final int DIRECTION_DEFAULT_LEFT_TO_RIGHT = -2;
76
77     /**
78      * Constant indicating that the base direction depends on the first strong
79      * directional character in the text according to the Unicode
80      * Bidirectional Algorithm. If no strong directional character is present,
81      * the base direction is right-to-left.
82      */
83     public static final int DIRECTION_DEFAULT_RIGHT_TO_LEFT = -1;
84
85     private BidiBase bidiBase;
86
87     /**
88      * Create Bidi from the given paragraph of text and base direction.
89      * @param paragraph a paragraph of text

```

```

90     * @param flags a collection of flags that control the algorithm. The
91     * algorithm understands the flags DIRECTION_LEFT_TO_RIGHT, DIRECTION_RIGHT_TO_LEFT,
92     * DIRECTION_DEFAULT_LEFT_TO_RIGHT, and DIRECTION_DEFAULT_RIGHT_TO_LEFT.
93     * Other values are reserved.
94     */
95     public Bidi(String paragraph, int flags) {
96         if (paragraph == null) {
97             throw new IllegalArgumentException("paragraph is null");
98         }
99
100         bidiBase = new BidiBase(paragraph.toCharArray(), 0, null, 0, paragraph.length(), flags);
101     }
102
103     /**
104     * Create Bidi from the given paragraph of text.
105     * <p>
106     * The RUN_DIRECTION attribute in the text, if present, determines the base
107     * direction (left-to-right or right-to-left). If not present, the base
108     * direction is computed using the Unicode Bidirectional Algorithm, defaulting to left-to-right
109     * if there are no strong directional characters in the text. This attribute, if
110     * present, must be applied to all the text in the paragraph.
111     * <p>
112     * The BIDI_EMBEDDING attribute in the text, if present, represents embedding level
113     * information. Negative values from -1 to -62 indicate overrides at the absolute value
114     * of the level. Positive values from 1 to 62 indicate embeddings. Where values are
115     * zero or not defined, the base embedding level as determined by the base direction
116     * is assumed.
117     * <p>
118     * The NUMERIC_SHAPING attribute in the text, if present, converts European digits to
119     * other decimal digits before running the bidi algorithm. This attribute, if present,
120     * must be applied to all the text in the paragraph.
121     *
122     * @param paragraph a paragraph of text with optional character and paragraph attribute
123     *         information
124     *
125     * @see java.awt.font.TextAttribute#BIDI_EMBEDDING
126     * @see java.awt.font.TextAttribute#NUMERIC_SHAPING
127     * @see java.awt.font.TextAttribute#RUN_DIRECTION
128     */
129     public Bidi(AttributedCharacterIterator paragraph) {
130         if (paragraph == null) {
131             throw new IllegalArgumentException("paragraph is null");
132         }
133
134         bidiBase = new BidiBase(0, 0);
135         bidiBase.setPara(paragraph);
136     }
137
138     /**
139     * Create Bidi from the given text, embedding, and direction information.
140     * The embeddings array may be null. If present, the values represent embedding level
141     * information. Negative values from -1 to -61 indicate overrides at the absolute value
142     * of the level. Positive values from 1 to 61 indicate embeddings. Where values are

```

```

142     * zero, the base embedding level as determined by the base direction is assumed.
143     * @param text an array containing the paragraph of text to process.
144     * @param textStart the index into the text array of the start of the paragraph.
145     * @param embeddings an array containing embedding values for each character in the paragraph.
146     * This can be null, in which case it is assumed that there is no external embedding
        information.
147     * @param embStart the index into the embedding array of the start of the paragraph.
148     * @param paragraphLength the length of the paragraph in the text and embeddings arrays.
149     * @param flags a collection of flags that control the algorithm. The
150     * algorithm understands the flags DIRECTION_LEFT_TO_RIGHT, DIRECTION_RIGHT_TO_LEFT,
151     * DIRECTION_DEFAULT_LEFT_TO_RIGHT, and DIRECTION_DEFAULT_RIGHT_TO_LEFT.
152     * Other values are reserved.
153     */
154     public Bidi(char[] text, int textStart, byte[] embeddings, int embStart, int paragraphLength,
        int flags) {
155         if (text == null) {
156             throw new IllegalArgumentException("text is null");
157         }
158         if (paragraphLength < 0) {
159             throw new IllegalArgumentException("bad length: " + paragraphLength);
160         }
161         if (textStart < 0 || paragraphLength > text.length - textStart) {
162             throw new IllegalArgumentException("bad range: " + textStart +
163                 " length: " + paragraphLength +
164                 " for text of length: " + text.length);
165         }
166         if (embeddings != null && (embStart < 0 || paragraphLength > embeddings.length - embStart))
            {
167             throw new IllegalArgumentException("bad range: " + embStart +
168                 " length: " + paragraphLength +
169                 " for embeddings of length: " + text.length);
170         }
171
172         bidiBase = new BidiBase(text, textStart, embeddings, embStart, paragraphLength, flags);
173     }
174
175     /**
176     * Create a Bidi object representing the bidi information on a line of text within
177     * the paragraph represented by the current Bidi. This call is not required if the
178     * entire paragraph fits on one line.
179     *
180     * @param lineStart the offset from the start of the paragraph to the start of the line.
181     * @param lineLimit the offset from the start of the paragraph to the limit of the line.
182     * @return a {@code Bidi} object
183     */
184     public Bidi createLineBidi(int lineStart, int lineLimit) {
185         AttributedString astr = new AttributedString("");
186         Bidi newBidi = new Bidi(astr.getIterator());
187
188         return bidiBase.setLine(this, bidiBase, newBidi, newBidi.bidiBase, lineStart, lineLimit);
189     }
190
191     /**

```

```
192     * Return true if the line is not left-to-right or right-to-left. This means it either has
193     * mixed runs of left-to-right
194     * and right-to-left text, or the base direction differs from the direction of the only run of
195     * text.
196     *
197     * @return true if the line is not left-to-right or right-to-left.
198     */
199     public boolean isMixed() {
200         return bidiBase.isMixed();
201     }
202
203     /**
204     * Return true if the line is all left-to-right text and the base direction is left-to-right.
205     *
206     * @return true if the line is all left-to-right text and the base direction is left-to-right
207     */
208     public boolean isLeftToRight() {
209         return bidiBase.isLeftToRight();
210     }
211
212     /**
213     * Return true if the line is all right-to-left text, and the base direction is right-to-left.
214     *
215     * @return true if the line is all right-to-left text, and the base direction is right-to-left
216     */
217     public boolean isRightToLeft() {
218         return bidiBase.isRightToLeft();
219     }
220
221     /**
222     * Return the length of text in the line.
223     *
224     * @return the length of text in the line
225     */
226     public int getLength() {
227         return bidiBase.getLength();
228     }
229
230     /**
231     * Return true if the base direction is left-to-right.
232     *
233     * @return true if the base direction is left-to-right
234     */
235     public boolean baseIsLeftToRight() {
236         return bidiBase.baseIsLeftToRight();
237     }
238
239     /**
240     * Return the base level (0 if left-to-right, 1 if right-to-left).
241     *
242     * @return the base level
243     */
244     public int getBaseLevel() {
245         return bidiBase.getParaLevel();
246     }
247
248     /**
```

```

243     * Return the resolved level of the character at offset.  If offset is
244     * {@literal <} 0 or &ge; the length of the line, return the base direction
245     * level.
246     *
247     * @param offset the index of the character for which to return the level
248     * @return the resolved level of the character at offset
249     */
250     public int getLevelAt(int offset) {
251         return bidiBase.getLevelAt(offset);
252     }
253
254     /**
255     * Return the number of level runs.
256     * @return the number of level runs
257     */
258     public int getRunCount() {
259         return bidiBase.countRuns();
260     }
261
262     /**
263     * Return the level of the nth logical run in this line.
264     * @param run the index of the run, between 0 and <code>getRunCount()</code>
265     * @return the level of the run
266     */
267     public int getRunLevel(int run) {
268         return bidiBase.getRunLevel(run);
269     }
270
271     /**
272     * Return the index of the character at the start of the nth logical run in this line, as
273     * an offset from the start of the line.
274     * @param run the index of the run, between 0 and <code>getRunCount()</code>
275     * @return the start of the run
276     */
277     public int getRunStart(int run) {
278         return bidiBase.getRunStart(run);
279     }
280
281     /**
282     * Return the index of the character past the end of the nth logical run in this line, as
283     * an offset from the start of the line.  For example, this will return the length
284     * of the line for the last run on the line.
285     * @param run the index of the run, between 0 and <code>getRunCount()</code>
286     * @return limit the limit of the run
287     */
288     public int getRunLimit(int run) {
289         return bidiBase.getRunLimit(run);
290     }
291
292     /**
293     * Return true if the specified text requires bidi analysis.  If this returns false,
294     * the text will display left-to-right.  Clients can then avoid constructing a Bidi object.
295     * Text in the Arabic Presentation Forms area of Unicode is presumed to already be shaped

```

```

296     * and ordered for display, and so will not cause this function to return true.
297     *
298     * @param text the text containing the characters to test
299     * @param start the start of the range of characters to test
300     * @param limit the limit of the range of characters to test
301     * @return true if the range of characters requires bidi analysis
302     */
303     public static boolean requiresBidi(char[] text, int start, int limit) {
304         return BidiBase.requiresBidi(text, start, limit);
305     }
306
307     /**
308     * Reorder the objects in the array into visual order based on their levels.
309     * This is a utility function to use when you have a collection of objects
310     * representing runs of text in logical order, each run containing text
311     * at a single level. The elements at <code>index</code> from
312     * <code>objectStart</code> up to <code>objectStart + count</code>
313     * in the objects array will be reordered into visual order assuming
314     * each run of text has the level indicated by the corresponding element
315     * in the levels array (at <code>index - objectStart + levelStart</code>).
316     *
317     * @param levels an array representing the bidi level of each object
318     * @param levelStart the start position in the levels array
319     * @param objects the array of objects to be reordered into visual order
320     * @param objectStart the start position in the objects array
321     * @param count the number of objects to reorder
322     */
323     public static void reorderVisually(byte[] levels, int levelStart, Object[] objects, int
        objectStart, int count) {
324         BidiBase.reorderVisually(levels, levelStart, objects, objectStart, count);
325     }
326
327     /**
328     * Display the bidi internal state, used in debugging.
329     */
330     public String toString() {
331         return bidiBase.toString();
332     }
333
334 }

```

2.5 BreakIterator.java

Secuencia 14: java/text/BreakIterator.java

```

1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *

```

```
10  *
11  *
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27   * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28   * (C) Copyright IBM Corp. 1996 - 1998 - All Rights Reserved
29   *
30   * The original version of this source code and documentation
31   * is copyrighted and owned by Taligent, Inc., a wholly-owned
32   * subsidiary of IBM. These materials are provided under terms
33   * of a License Agreement between Taligent and Sun. This technology
34   * is protected by multiple US and International patents.
35   *
36   * This notice and attribution to Taligent may not be removed.
37   * Taligent is a registered trademark of Taligent, Inc.
38   *
39   */
40
41  package java.text;
42
43  import java.lang.ref.SoftReference;
44  import java.text.spi.BreakIteratorProvider;
45  import java.util.Locale;
46  import sun.util.locale.provider.LocaleProviderAdapter;
47  import sun.util.locale.provider.LocaleServiceProviderPool;
48
49
50  /**
51   * The BreakIterator class implements methods for finding
52   * the location of boundaries in text. Instances of BreakIterator
53   * maintain a current position and scan over text
54   * returning the index of characters where boundaries occur.
55   * Internally, BreakIterator scans text using a
56   * CharacterIterator, and is thus able to scan text held
57   * by any object implementing that protocol. A StringCharacterIterator
58   * is used to scan String objects passed to setText.
59   *
60   * <p>
61   * You use the factory methods provided by this class to create
62   * instances of various types of break iterators. In particular,
```



```

63 * use <code>getWordInstance</code>, <code>getLineInstance</code>,
64 * <code>getSentenceInstance</code>, and <code>getCharacterInstance</code>
65 * to create <code>BreakIterator</code>s that perform
66 * word, line, sentence, and character boundary analysis respectively.
67 * A single <code>BreakIterator</code> can work only on one unit
68 * (word, line, sentence, and so on). You must use a different iterator
69 * for each unit boundary analysis you wish to perform.
70 *
71 * <p><a name="line"></a>
72 * Line boundary analysis determines where a text string can be
73 * broken when line-wrapping. The mechanism correctly handles
74 * punctuation and hyphenated words. Actual line breaking needs
75 * to also consider the available line width and is handled by
76 * higher-level software.
77 *
78 * <p><a name="sentence"></a>
79 * Sentence boundary analysis allows selection with correct interpretation
80 * of periods within numbers and abbreviations, and trailing punctuation
81 * marks such as quotation marks and parentheses.
82 *
83 * <p><a name="word"></a>
84 * Word boundary analysis is used by search and replace functions, as
85 * well as within text editing applications that allow the user to
86 * select words with a double click. Word selection provides correct
87 * interpretation of punctuation marks within and following
88 * words. Characters that are not part of a word, such as symbols
89 * or punctuation marks, have word-breaks on both sides.
90 *
91 * <p><a name="character"></a>
92 * Character boundary analysis allows users to interact with characters
93 * as they expect to, for example, when moving the cursor through a text
94 * string. Character boundary analysis provides correct navigation
95 * through character strings, regardless of how the character is stored.
96 * The boundaries returned may be those of supplementary characters,
97 * combining character sequences, or ligature clusters.
98 * For example, an accented character might be stored as a base character
99 * and a diacritical mark. What users consider to be a character can
100 * differ between languages.
101 *
102 * <p>
103 * The <code>BreakIterator</code> instances returned by the factory methods
104 * of this class are intended for use with natural languages only, not for
105 * programming language text. It is however possible to define subclasses
106 * that tokenize a programming language.
107 *
108 * <P>
109 * <strong>Examples</strong>:<P>
110 * Creating and using text boundaries:
111 * <blockquote>
112 * <pre>
113 * public static void main(String args[]) {
114 *     if (args.length == 1) {
115 *         String stringToExamine = args[0];

```

```
116 *      //print each word in order
117 *      BreakIterator boundary = BreakIterator.getWordInstance();
118 *      boundary.setText(stringToExamine);
119 *      printEachForward(boundary, stringToExamine);
120 *      //print each sentence in reverse order
121 *      boundary = BreakIterator.getSentenceInstance(Locale.US);
122 *      boundary.setText(stringToExamine);
123 *      printEachBackward(boundary, stringToExamine);
124 *      printFirst(boundary, stringToExamine);
125 *      printLast(boundary, stringToExamine);
126 *      }
127 * }
128 * </pre>
129 * </blockquote>
130 *
131 * Print each element in order:
132 * <blockquote>
133 * <pre>
134 * public static void printEachForward(BreakIterator boundary, String source) {
135 *     int start = boundary.first();
136 *     for (int end = boundary.next();
137 *         end != BreakIterator.DONE;
138 *         start = end, end = boundary.next()) {
139 *         System.out.println(source.substring(start,end));
140 *     }
141 * }
142 * </pre>
143 * </blockquote>
144 *
145 * Print each element in reverse order:
146 * <blockquote>
147 * <pre>
148 * public static void printEachBackward(BreakIterator boundary, String source) {
149 *     int end = boundary.last();
150 *     for (int start = boundary.previous();
151 *         start != BreakIterator.DONE;
152 *         end = start, start = boundary.previous()) {
153 *         System.out.println(source.substring(start,end));
154 *     }
155 * }
156 * </pre>
157 * </blockquote>
158 *
159 * Print first element:
160 * <blockquote>
161 * <pre>
162 * public static void printFirst(BreakIterator boundary, String source) {
163 *     int start = boundary.first();
164 *     int end = boundary.next();
165 *     System.out.println(source.substring(start,end));
166 * }
167 * </pre>
168 * </blockquote>
```

```

169 *
170 * Print last element:
171 * <blockquote>
172 * <pre>
173 * public static void printLast(BreakIterator boundary, String source) {
174 *     int end = boundary.last();
175 *     int start = boundary.previous();
176 *     System.out.println(source.substring(start,end));
177 * }
178 * </pre>
179 * </blockquote>
180 *
181 * Print the element at a specified position:
182 * <blockquote>
183 * <pre>
184 * public static void printAt(BreakIterator boundary, int pos, String source) {
185 *     int end = boundary.following(pos);
186 *     int start = boundary.previous();
187 *     System.out.println(source.substring(start,end));
188 * }
189 * </pre>
190 * </blockquote>
191 *
192 * Find the next word:
193 * <blockquote>
194 * <pre>{@code
195 * public static int nextWordStartAfter(int pos, String text) {
196 *     BreakIterator wb = BreakIterator.getWordInstance();
197 *     wb.setText(text);
198 *     int last = wb.following(pos);
199 *     int current = wb.next();
200 *     while (current != BreakIterator.DONE) {
201 *         for (int p = last; p < current; p++) {
202 *             if (Character.isLetter(text.codePointAt(p)))
203 *                 return last;
204 *         }
205 *         last = current;
206 *         current = wb.next();
207 *     }
208 *     return BreakIterator.DONE;
209 * }
210 * }</pre>
211 * (The iterator returned by BreakIterator.getWordInstance() is unique in that
212 * the break positions it returns don't represent both the start and end of the
213 * thing being iterated over. That is, a sentence-break iterator returns breaks
214 * that each represent the end of one sentence and the beginning of the next.
215 * With the word-break iterator, the characters between two boundaries might be a
216 * word, or they might be the punctuation or whitespace between two words. The
217 * above code uses a simple heuristic to determine which boundary is the beginning
218 * of a word: If the characters between this boundary and the next boundary
219 * include at least one letter (this can be an alphabetical letter, a CJK ideograph,
220 * a Hangul syllable, a Kana character, etc.), then the text between this boundary
221 * and the next is a word; otherwise, it's the material between words.)

```

```
222 * </blockquote>
223 *
224 * @see CharacterIterator
225 *
226 */
227
228 public abstract class BreakIterator implements Cloneable
229 {
230     /**
231      * Constructor. BreakIterator is stateless and has no default behavior.
232      */
233     protected BreakIterator()
234     {
235     }
236
237     /**
238      * Create a copy of this iterator
239      * @return A copy of this
240      */
241     @Override
242     public Object clone()
243     {
244         try {
245             return super.clone();
246         }
247         catch (CloneNotSupportedException e) {
248             throw new InternalError(e);
249         }
250     }
251
252     /**
253      * DONE is returned by previous(), next(), next(int), preceding(int)
254      * and following(int) when either the first or last text boundary has been
255      * reached.
256      */
257     public static final int DONE = -1;
258
259     /**
260      * Returns the first boundary. The iterator's current position is set
261      * to the first text boundary.
262      * @return The character index of the first text boundary.
263      */
264     public abstract int first();
265
266     /**
267      * Returns the last boundary. The iterator's current position is set
268      * to the last text boundary.
269      * @return The character index of the last text boundary.
270      */
271     public abstract int last();
272
273     /**
274      * Returns the nth boundary from the current boundary. If either
```

```

275     * the first or last text boundary has been reached, it returns
276     * BreakIterator.DONE and the current position is set to either
277     * the first or last text boundary depending on which one is reached. Otherwise,
278     * the iterator's current position is set to the new boundary.
279     * For example, if the iterator's current position is the mth text boundary
280     * and three more boundaries exist from the current boundary to the last text
281     * boundary, the next(2) call will return m + 2. The new text position is set
282     * to the (m + 2)th text boundary. A next(4) call would return
283     * BreakIterator.DONE and the last text boundary would become the
284     * new text position.
285     * @param n which boundary to return. A value of 0
286     * does nothing. Negative values move to previous boundaries
287     * and positive values move to later boundaries.
288     * @return The character index of the nth boundary from the current position
289     * or BreakIterator.DONE if either first or last text boundary
290     * has been reached.
291     */
292     public abstract int next(int n);
293
294     /**
295     * Returns the boundary following the current boundary. If the current boundary
296     * is the last text boundary, it returns BreakIterator.DONE and
297     * the iterator's current position is unchanged. Otherwise, the iterator's
298     * current position is set to the boundary following the current boundary.
299     * @return The character index of the next text boundary or
300     * BreakIterator.DONE if the current boundary is the last text
301     * boundary.
302     * Equivalent to next(1).
303     * @see #next(int)
304     */
305     public abstract int next();
306
307     /**
308     * Returns the boundary preceding the current boundary. If the current boundary
309     * is the first text boundary, it returns BreakIterator.DONE and
310     * the iterator's current position is unchanged. Otherwise, the iterator's
311     * current position is set to the boundary preceding the current boundary.
312     * @return The character index of the previous text boundary or
313     * BreakIterator.DONE if the current boundary is the first text
314     * boundary.
315     */
316     public abstract int previous();
317
318     /**
319     * Returns the first boundary following the specified character offset. If the
320     * specified offset equals to the last text boundary, it returns
321     * BreakIterator.DONE and the iterator's current position is unchanged.
322     * Otherwise, the iterator's current position is set to the returned boundary.
323     * The value returned is always greater than the offset or the value
324     * BreakIterator.DONE.
325     * @param offset the character offset to begin scanning.
326     * @return The first boundary after the specified offset or
327     * BreakIterator.DONE if the last text boundary is passed in

```

```

328     * as the offset.
329     * @exception IllegalArgumentException if the specified offset is less than
330     * the first text boundary or greater than the last text boundary.
331     */
332     public abstract int following(int offset);
333
334     /**
335     * Returns the last boundary preceding the specified character offset. If the
336     * specified offset equals to the first text boundary, it returns
337     * BreakIterator.DONE and the iterator's current position is unchanged.
338     * Otherwise, the iterator's current position is set to the returned boundary.
339     * The value returned is always less than the offset or the value
340     * BreakIterator.DONE.
341     * @param offset the character offset to begin scanning.
342     * @return The last boundary before the specified offset or
343     * BreakIterator.DONE if the first text boundary is passed in
344     * as the offset.
345     * @exception IllegalArgumentException if the specified offset is less than
346     * the first text boundary or greater than the last text boundary.
347     * @since 1.2
348     */
349     public int preceding(int offset) {
350         // NOTE: This implementation is here solely because we can't add new
351         // abstract methods to an existing class. There is almost ALWAYS a
352         // better, faster way to do this.
353         int pos = following(offset);
354         while (pos >= offset && pos != DONE) {
355             pos = previous();
356         }
357         return pos;
358     }
359
360     /**
361     * Returns true if the specified character offset is a text boundary.
362     * @param offset the character offset to check.
363     * @return true if "offset" is a boundary position,
364     * false otherwise.
365     * @exception IllegalArgumentException if the specified offset is less than
366     * the first text boundary or greater than the last text boundary.
367     * @since 1.2
368     */
369     public boolean isBoundary(int offset) {
370         // NOTE: This implementation probably is wrong for most situations
371         // because it fails to take into account the possibility that a
372         // CharacterIterator passed to setText() may not have a begin offset
373         // of 0. But since the abstract BreakIterator doesn't have that
374         // knowledge, it assumes the begin offset is 0. If you subclass
375         // BreakIterator, copy the SimpleTextBoundary implementation of this
376         // function into your subclass. [This should have been abstract at
377         // this level, but it's too late to fix that now.]
378         if (offset == 0) {
379             return true;
380         }

```

```
381     int boundary = following(offset - 1);
382     if (boundary == DONE) {
383         throw new IllegalArgumentException();
384     }
385     return boundary == offset;
386 }
387
388 /**
389  * Returns character index of the text boundary that was most
390  * recently returned by next(), next(int), previous(), first(), last(),
391  * following(int) or preceding(int). If any of these methods returns
392  * <code>BreakIterator.DONE</code> because either first or last text boundary
393  * has been reached, it returns the first or last text boundary depending on
394  * which one is reached.
395  * @return The text boundary returned from the above methods, first or last
396  * text boundary.
397  * @see #next()
398  * @see #next(int)
399  * @see #previous()
400  * @see #first()
401  * @see #last()
402  * @see #following(int)
403  * @see #preceding(int)
404  */
405 public abstract int current();
406
407 /**
408  * Get the text being scanned
409  * @return the text being scanned
410  */
411 public abstract CharacterIterator getText();
412
413 /**
414  * Set a new text string to be scanned. The current scan
415  * position is reset to first().
416  * @param newText new text to scan.
417  */
418 public void setText(String newText)
419 {
420     setText(new StringCharacterIterator(newText));
421 }
422
423 /**
424  * Set a new text for scanning. The current scan
425  * position is reset to first().
426  * @param newText new text to scan.
427  */
428 public abstract void setText(CharacterIterator newText);
429
430 private static final int CHARACTER_INDEX = 0;
431 private static final int WORD_INDEX = 1;
432 private static final int LINE_INDEX = 2;
433 private static final int SENTENCE_INDEX = 3;
```

```
434
435 @SuppressWarnings("unchecked")
436 private static final SoftReference<BreakIteratorCache>[] iterCache = (SoftReference<
    BreakIteratorCache>[]) new SoftReference<?>[4];
437
438 /**
439  * Returns a new <code>BreakIterator</code> instance
440  * for <a href="BreakIterator.html#word">word breaks</a>
441  * for the {@linkplain Locale#getDefault() default locale}.
442  * @return A break iterator for word breaks
443  */
444 public static BreakIterator getWordInstance()
445 {
446     return getWordInstance(Locale.getDefault());
447 }
448
449 /**
450  * Returns a new <code>BreakIterator</code> instance
451  * for <a href="BreakIterator.html#word">word breaks</a>
452  * for the given locale.
453  * @param locale the desired locale
454  * @return A break iterator for word breaks
455  * @exception NullPointerException if <code>locale</code> is null
456  */
457 public static BreakIterator getWordInstance(Locale locale)
458 {
459     return getBreakInstance(locale, WORD_INDEX);
460 }
461
462 /**
463  * Returns a new <code>BreakIterator</code> instance
464  * for <a href="BreakIterator.html#line">line breaks</a>
465  * for the {@linkplain Locale#getDefault() default locale}.
466  * @return A break iterator for line breaks
467  */
468 public static BreakIterator getLineInstance()
469 {
470     return getLineInstance(Locale.getDefault());
471 }
472
473 /**
474  * Returns a new <code>BreakIterator</code> instance
475  * for <a href="BreakIterator.html#line">line breaks</a>
476  * for the given locale.
477  * @param locale the desired locale
478  * @return A break iterator for line breaks
479  * @exception NullPointerException if <code>locale</code> is null
480  */
481 public static BreakIterator getLineInstance(Locale locale)
482 {
483     return getBreakInstance(locale, LINE_INDEX);
484 }
485
```



```

486 /**
487  * Returns a new <code>BreakIterator</code> instance
488  * for <a href="BreakIterator.html#character">character breaks</a>
489  * for the {@linkplain Locale#getDefault() default locale}.
490  * @return A break iterator for character breaks
491  */
492 public static BreakIterator getCharacterInstance()
493 {
494     return getCharacterInstance(Locale.getDefault());
495 }
496
497 /**
498  * Returns a new <code>BreakIterator</code> instance
499  * for <a href="BreakIterator.html#character">character breaks</a>
500  * for the given locale.
501  * @param locale the desired locale
502  * @return A break iterator for character breaks
503  * @exception NullPointerException if <code>locale</code> is null
504  */
505 public static BreakIterator getCharacterInstance(Locale locale)
506 {
507     return getBreakInstance(locale, CHARACTER_INDEX);
508 }
509
510 /**
511  * Returns a new <code>BreakIterator</code> instance
512  * for <a href="BreakIterator.html#sentence">sentence breaks</a>
513  * for the {@linkplain Locale#getDefault() default locale}.
514  * @return A break iterator for sentence breaks
515  */
516 public static BreakIterator getSentenceInstance()
517 {
518     return getSentenceInstance(Locale.getDefault());
519 }
520
521 /**
522  * Returns a new <code>BreakIterator</code> instance
523  * for <a href="BreakIterator.html#sentence">sentence breaks</a>
524  * for the given locale.
525  * @param locale the desired locale
526  * @return A break iterator for sentence breaks
527  * @exception NullPointerException if <code>locale</code> is null
528  */
529 public static BreakIterator getSentenceInstance(Locale locale)
530 {
531     return getBreakInstance(locale, SENTENCE_INDEX);
532 }
533
534 private static BreakIterator getBreakInstance(Locale locale, int type) {
535     if (iterCache[type] != null) {
536         BreakIteratorCache cache = iterCache[type].get();
537         if (cache != null) {
538             if (cache.getLocale().equals(locale)) {

```

```

539         return cache.createBreakInstance();
540     }
541 }
542 }
543
544 BreakIterator result = createBreakInstance(locale, type);
545 BreakIteratorCache cache = new BreakIteratorCache(locale, result);
546 iterCache[type] = new SoftReference<>(cache);
547 return result;
548 }
549
550 private static BreakIterator createBreakInstance(Locale locale,
551                                             int type) {
552     LocaleProviderAdapter adapter = LocaleProviderAdapter.getAdapter(BreakIteratorProvider.
553                             class, locale);
554     BreakIterator iterator = createBreakInstance(adapter, locale, type);
555     if (iterator == null) {
556         iterator = createBreakInstance(LocaleProviderAdapter.forJRE(), locale, type);
557     }
558     return iterator;
559 }
560
561 private static BreakIterator createBreakInstance(LocaleProviderAdapter adapter, Locale locale,
562                                             int type) {
563     BreakIteratorProvider breakIteratorProvider = adapter.getBreakIteratorProvider();
564     BreakIterator iterator = null;
565     switch (type) {
566     case CHARACTER_INDEX:
567         iterator = breakIteratorProvider.getCharacterInstance(locale);
568         break;
569     case WORD_INDEX:
570         iterator = breakIteratorProvider.getWordInstance(locale);
571         break;
572     case LINE_INDEX:
573         iterator = breakIteratorProvider.getLineInstance(locale);
574         break;
575     case SENTENCE_INDEX:
576         iterator = breakIteratorProvider.getSentenceInstance(locale);
577         break;
578     }
579     return iterator;
580 }
581
582 /**
583  * Returns an array of all locales for which the
584  * <code>get*Instance</code> methods of this class can return
585  * localized instances.
586  * The returned array represents the union of locales supported by the Java
587  * runtime and by installed
588  * {@link java.text.spi.BreakIteratorProvider BreakIteratorProvider} implementations.
589  * It must contain at least a <code>Locale</code>
590  * instance equal to {@link java.util.Locale#US Locale.US}.
591  */

```

```

590     * @return An array of locales for which localized
591     *         <code>BreakIterator</code> instances are available.
592     */
593     public static synchronized Locale[] getAvailableLocales()
594     {
595         LocaleServiceProviderPool pool =
596             LocaleServiceProviderPool.getPool(BreakIteratorProvider.class);
597         return pool.getAvailableLocales();
598     }
599
600     private static final class BreakIteratorCache {
601
602         private BreakIterator iter;
603         private Locale locale;
604
605         BreakIteratorCache(Locale locale, BreakIterator iter) {
606             this.locale = locale;
607             this.iter = (BreakIterator) iter.clone();
608         }
609
610         Locale getLocale() {
611             return locale;
612         }
613
614         BreakIterator createBreakInstance() {
615             return (BreakIterator) iter.clone();
616         }
617     }
618 }

```

2.6 CalendarBuilder.java

Secuencia 15: java/text/CalendarBuilder.java

```

1  /*
2  * Copyright (c) 2010, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *

```

```

21  *
22  *
23  *
24  */
25
26  package java.text;
27
28  import java.util.Calendar;
29  import static java.util.GregorianCalendar.*;
30
31  /**
32   * {@code CalendarBuilder} keeps field-value pairs for setting
33   * the calendar fields of the given {@code Calendar}. It has the
34   * {@code Calendar#FIELD_COUNT FIELD_COUNT}-th field for the week year
35   * support. Also {@code ISO_DAY_OF_WEEK} is used to specify
36   * {@code DAY_OF_WEEK} in the ISO day of week numbering.
37   *
38   * <p>{@code CalendarBuilder} retains the semantic of the pseudo
39   * timestamp for fields. {@code CalendarBuilder} uses a single
40   * int array combining fields[] and stamp[] of {@code Calendar}.
41   *
42   * @author Masayoshi Okutsu
43   */
44  class CalendarBuilder {
45      /*
46       * Pseudo time stamp constants used in java.util.Calendar
47       */
48      private static final int UNSET = 0;
49      private static final int COMPUTED = 1;
50      private static final int MINIMUM_USER_STAMP = 2;
51
52      private static final int MAX_FIELD = FIELD_COUNT + 1;
53
54      public static final int WEEK_YEAR = FIELD_COUNT;
55      public static final int ISO_DAY_OF_WEEK = 1000; // pseudo field index
56
57      // stamp[] (lower half) and field[] (upper half) combined
58      private final int[] field;
59      private int nextStamp;
60      private int maxFieldIndex;
61
62      CalendarBuilder() {
63          field = new int[MAX_FIELD * 2];
64          nextStamp = MINIMUM_USER_STAMP;
65          maxFieldIndex = -1;
66      }
67
68      CalendarBuilder set(int index, int value) {
69          if (index == ISO_DAY_OF_WEEK) {
70              index = DAY_OF_WEEK;
71              value = toCalendarDayOfWeek(value);
72          }
73          field[index] = nextStamp++;

```

```

74     field[MAX_FIELD + index] = value;
75     if (index > maxFieldIndex && index < FIELD_COUNT) {
76         maxFieldIndex = index;
77     }
78     return this;
79 }
80
81 CalendarBuilder addYear(int value) {
82     field[MAX_FIELD + YEAR] += value;
83     field[MAX_FIELD + WEEK_YEAR] += value;
84     return this;
85 }
86
87 boolean isSet(int index) {
88     if (index == ISO_DAY_OF_WEEK) {
89         index = DAY_OF_WEEK;
90     }
91     return field[index] > UNSET;
92 }
93
94 CalendarBuilder clear(int index) {
95     if (index == ISO_DAY_OF_WEEK) {
96         index = DAY_OF_WEEK;
97     }
98     field[index] = UNSET;
99     field[MAX_FIELD + index] = 0;
100    return this;
101 }
102
103 Calendar establish(Calendar cal) {
104     boolean weekDate = isSet(WEEK_YEAR)
105         && field[WEEK_YEAR] > field[YEAR];
106     if (weekDate && !cal.isWeekDateSupported()) {
107         // Use YEAR instead
108         if (!isSet(YEAR)) {
109             set(YEAR, field[MAX_FIELD + WEEK_YEAR]);
110         }
111         weekDate = false;
112     }
113
114     cal.clear();
115     // Set the fields from the min stamp to the max stamp so that
116     // the field resolution works in the Calendar.
117     for (int stamp = MINIMUM_USER_STAMP; stamp < nextStamp; stamp++) {
118         for (int index = 0; index <= maxFieldIndex; index++) {
119             if (field[index] == stamp) {
120                 cal.set(index, field[MAX_FIELD + index]);
121                 break;
122             }
123         }
124     }
125
126     if (weekDate) {

```

```

127     int weekOfYear = isSet(WEEK_OF_YEAR) ? field[MAX_FIELD + WEEK_OF_YEAR] : 1;
128     int dayOfWeek = isSet(DAY_OF_WEEK) ?
129         field[MAX_FIELD + DAY_OF_WEEK] : cal.getFirstDayOfWeek();
130     if (!isValidDayOfWeek(dayOfWeek) && cal.isLenient()) {
131         if (dayOfWeek >= 8) {
132             dayOfWeek--;
133             weekOfYear += dayOfWeek / 7;
134             dayOfWeek = (dayOfWeek % 7) + 1;
135         } else {
136             while (dayOfWeek <= 0) {
137                 dayOfWeek += 7;
138                 weekOfYear--;
139             }
140         }
141         dayOfWeek = toCalendarDayOfWeek(dayOfWeek);
142     }
143     cal.setWeekDate(field[MAX_FIELD + WEEK_YEAR], weekOfYear, dayOfWeek);
144 }
145 return cal;
146 }
147
148 public String toString() {
149     StringBuilder sb = new StringBuilder();
150     sb.append("CalendarBuilder:");
151     for (int i = 0; i < field.length; i++) {
152         if (isSet(i)) {
153             sb.append(i).append('=').append(field[MAX_FIELD + i]).append(',');
154         }
155     }
156     int lastIndex = sb.length() - 1;
157     if (sb.charAt(lastIndex) == ',') {
158         sb.setLength(lastIndex);
159     }
160     sb.append(']');
161     return sb.toString();
162 }
163
164 static int toISODayOfWeek(int calendarDayOfWeek) {
165     return calendarDayOfWeek == SUNDAY ? 7 : calendarDayOfWeek - 1;
166 }
167
168 static int toCalendarDayOfWeek(int isoDayOfWeek) {
169     if (!isValidDayOfWeek(isoDayOfWeek)) {
170         // adjust later for lenient mode
171         return isoDayOfWeek;
172     }
173     return isoDayOfWeek == 7 ? SUNDAY : isoDayOfWeek + 1;
174 }
175
176 static boolean isValidDayOfWeek(int dayOfWeek) {
177     return dayOfWeek > 0 && dayOfWeek <= 7;
178 }
179 }

```

2.7 CharacterIterator.java

Secuencia 16: java/text/CharacterIterator.java

```
1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28 * (C) Copyright IBM Corp. 1996 - 1998 - All Rights Reserved
29 *
30 * The original version of this source code and documentation
31 * is copyrighted and owned by Taligent, Inc., a wholly-owned
32 * subsidiary of IBM. These materials are provided under terms
33 * of a License Agreement between Taligent and Sun. This technology
34 * is protected by multiple US and International patents.
35 *
36 * This notice and attribution to Taligent may not be removed.
37 * Taligent is a registered trademark of Taligent, Inc.
38 *
39 */
40
41 package java.text;
42
43
44 /**
45 * This interface defines a protocol for bidirectional iteration over text.
46 * The iterator iterates over a bounded sequence of characters. Characters
47 * are indexed with values beginning with the value returned by getBeginIndex() and
48 * continuing through the value returned by getEndIndex()-1.
49 * <p>
50 * Iterators maintain a current character index, whose valid range is from
```

```

51 * getBeginIndex() to getEndIndex(); the value getEndIndex() is included to allow
52 * handling of zero-length text ranges and for historical reasons.
53 * The current index can be retrieved by calling getIndex() and set directly
54 * by calling setIndex(), first(), and last().
55 * <p>
56 * The methods previous() and next() are used for iteration. They return DONE if
57 * they would move outside the range from getBeginIndex() to getEndIndex() -1,
58 * signaling that the iterator has reached the end of the sequence. DONE is
59 * also returned by other methods to indicate that the current index is
60 * outside this range.
61 *
62 * <P>Examples:<P>
63 *
64 * Traverse the text from start to finish
65 * <pre>{@code
66 * public void traverseForward(CharacterIterator iter) {
67 *     for(char c = iter.first(); c != CharacterIterator.DONE; c = iter.next()) {
68 *         processChar(c);
69 *     }
70 * }
71 * }</pre>
72 *
73 * Traverse the text backwards, from end to start
74 * <pre>{@code
75 * public void traverseBackward(CharacterIterator iter) {
76 *     for(char c = iter.last(); c != CharacterIterator.DONE; c = iter.previous()) {
77 *         processChar(c);
78 *     }
79 * }
80 * }</pre>
81 *
82 * Traverse both forward and backward from a given position in the text.
83 * Calls to notBoundary() in this example represents some
84 * additional stopping criteria.
85 * <pre>{@code
86 * public void traverseOut(CharacterIterator iter, int pos) {
87 *     for (char c = iter.setIndex(pos);
88 *         c != CharacterIterator.DONE && notBoundary(c);
89 *         c = iter.next()) {
90 *     }
91 *     int end = iter.getIndex();
92 *     for (char c = iter.setIndex(pos);
93 *         c != CharacterIterator.DONE && notBoundary(c);
94 *         c = iter.previous()) {
95 *     }
96 *     int start = iter.getIndex();
97 *     processSection(start, end);
98 * }
99 * }</pre>
100 *
101 * @see StringCharacterIterator
102 * @see AttributedCharacterIterator
103 */

```



```
104
105 public interface CharacterIterator extends Cloneable
106 {
107
108     /**
109      * Constant that is returned when the iterator has reached either the end
110      * or the beginning of the text. The value is '\\uFFFF', the "not a
111      * character" value which should not occur in any valid Unicode string.
112      */
113     public static final char DONE = '\\uFFFF';
114
115     /**
116      * Sets the position to getBeginIndex() and returns the character at that
117      * position.
118      * @return the first character in the text, or DONE if the text is empty
119      * @see #getBeginIndex()
120      */
121     public char first();
122
123     /**
124      * Sets the position to getEndIndex()-1 (getEndIndex() if the text is empty)
125      * and returns the character at that position.
126      * @return the last character in the text, or DONE if the text is empty
127      * @see #getEndIndex()
128      */
129     public char last();
130
131     /**
132      * Gets the character at the current position (as returned by getIndex()).
133      * @return the character at the current position or DONE if the current
134      * position is off the end of the text.
135      * @see #getIndex()
136      */
137     public char current();
138
139     /**
140      * Increments the iterator's index by one and returns the character
141      * at the new index. If the resulting index is greater or equal
142      * to getEndIndex(), the current index is reset to getEndIndex() and
143      * a value of DONE is returned.
144      * @return the character at the new position or DONE if the new
145      * position is off the end of the text range.
146      */
147     public char next();
148
149     /**
150      * Decrements the iterator's index by one and returns the character
151      * at the new index. If the current index is getBeginIndex(), the index
152      * remains at getBeginIndex() and a value of DONE is returned.
153      * @return the character at the new position or DONE if the current
154      * position is equal to getBeginIndex().
155      */
156     public char previous();
```

```

157
158 /**
159  * Sets the position to the specified position in the text and returns that
160  * character.
161  * @param position the position within the text. Valid values range from
162  * getBeginIndex() to getEndIndex(). An IllegalArgumentException is thrown
163  * if an invalid value is supplied.
164  * @return the character at the specified position or DONE if the specified position is equal
165  *         to getEndIndex()
166  */
167 public char setIndex(int position);
168
169 /**
170  * Returns the start index of the text.
171  * @return the index at which the text begins.
172  */
173 public int getBeginIndex();
174
175 /**
176  * Returns the end index of the text. This index is the index of the first
177  * character following the end of the text.
178  * @return the index after the last character in the text
179  */
180 public int getEndIndex();
181
182 /**
183  * Returns the current index.
184  * @return the current index.
185  */
186 public int getIndex();
187
188 /**
189  * Create a copy of this iterator
190  * @return A copy of this
191  */
192 public Object clone();
193 }

```

2.8 CharacterIteratorFieldDelegate.java

Secuencia 17: java/text/CharacterIteratorFieldDelegate.java

```

1  /*
2  * Copyright (c) 2000, 2012, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *

```

```
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25  package java.text;
26
27  import java.util.ArrayList;
28
29  /**
30   * CharacterIteratorFieldDelegate combines the notifications from a Format
31   * into a resulting AttributedCharacterIterator. The resulting
32   * AttributedCharacterIterator can be retrieved by way of
33   * the getIterator method.
34   *
35   */
36  class CharacterIteratorFieldDelegate implements Format.FieldDelegate {
37      /**
38       * Array of AttributeStrings. Whenever formatted is invoked
39       * for a region > size, a new instance of AttributedString is added to
40       * attributedStrings. Subsequent invocations of formatted
41       * for existing regions result in invoking addAttribute on the existing
42       * AttributedString.
43       */
44      private ArrayList<AttributedString> attributedStrings;
45      /**
46       * Running count of the number of characters that have
47       * been encountered.
48       */
49      private int size;
50
51
52      CharacterIteratorFieldDelegate() {
53          attributedStrings = new ArrayList<>();
54      }
55
56      public void formatted(Format.Field attr, Object value, int start, int end,
57                          StringBuffer buffer) {
58          if (start != end) {
59              if (start < size) {
60                  // Adjust attributes of existing runs
61                  int index = size;
62                  int asIndex = attributedStrings.size() - 1;
63
64                  while (start < index) {
```

```

65         AttributedString as = attributedStrings.
66             get(asIndex--);
67         int newIndex = index - as.length();
68         int aStart = Math.max(0, start - newIndex);
69
70         as.addAttribute(attr, value, aStart, Math.min(
71             end - start, as.length() - aStart) +
72             aStart);
73         index = newIndex;
74     }
75 }
76 if (size < start) {
77     // Pad attributes
78     attributedStrings.add(new AttributedString(
79         buffer.substring(size, start)));
80     size = start;
81 }
82 if (size < end) {
83     // Add new string
84     int aStart = Math.max(start, size);
85     AttributedString string = new AttributedString(
86         buffer.substring(aStart, end));
87
88     string.addAttribute(attr, value);
89     attributedStrings.add(string);
90     size = end;
91 }
92 }
93 }
94
95 public void formatted(int fieldID, Format.Field attr, Object value,
96     int start, int end, StringBuffer buffer) {
97     formatted(attr, value, start, end, buffer);
98 }
99
100 /**
101  * Returns an AttributedString that can be used
102  * to iterate over the resulting formatted String.
103  *
104  * @param string Result of formatting.
105  */
106 public AttributedString getIterator(String string) {
107     // Add the last AttributedString if necessary
108     // assert(size <= string.length());
109     if (string.length() > size) {
110         attributedStrings.add(new AttributedString(
111             string.substring(size)));
112         size = string.length();
113     }
114     int iCount = attributedStrings.size();
115     AttributedString iterators[] = new
116         AttributedString[iCount];
117

```

```

118     for (int counter = 0; counter < iCount; counter++) {
119         iterators[counter] = attributedStrings.
120             get(counter).getIterator();
121     }
122     return new AttributedString(iterators).getIterator();
123 }
124 }

```

2.9 ChoiceFormat.java

Secuencia 18: java/text/ChoiceFormat.java

```

1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28 * (C) Copyright IBM Corp. 1996 - 1998 - All Rights Reserved
29 *
30 * The original version of this source code and documentation is copyrighted
31 * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32 * materials are provided under terms of a License Agreement between Taligent
33 * and Sun. This technology is protected by multiple US and International
34 * patents. This notice and attribution to Taligent may not be removed.
35 * Taligent is a registered trademark of Taligent, Inc.
36 *
37 */
38
39 package java.text;
40
41 import java.io.InvalidObjectException;
42 import java.io.IOException;

```

```

43 import java.io.ObjectInputStream;
44 import java.util.Arrays;
45
46 /**
47  * A ChoiceFormat allows you to attach a format to a range of numbers.
48  * It is generally used in a MessageFormat for handling plurals.
49  * The choice is specified with an ascending list of doubles, where each item
50  * specifies a half-open interval up to the next item:
51  * <blockquote>
52  * <pre>
53  * X matches j if and only if limit[j] &lt;= X &lt; limit[j+1]
54  * </pre>
55  * </blockquote>
56  * If there is no match, then either the first or last index is used, depending
57  * on whether the number (X) is too low or too high. If the limit array is not
58  * in ascending order, the results of formatting will be incorrect. ChoiceFormat
59  * also accepts &#92;u221E as equivalent to infinity(INF).
60  *
61  * <p>
62  * <strong>Note:
63  * <code>ChoiceFormat differs from the other <code>Format
64  * classes in that you create a <code>ChoiceFormat object with a
65  * constructor (not with a <code>getInstance style factory
66  * method). The factory methods aren't necessary because <code>ChoiceFormat
67  * doesn't require any complex setup for a given locale. In fact,
68  * <code>ChoiceFormat doesn't implement any locale specific behavior.
69  *
70  * <p>
71  * When creating a <code>ChoiceFormat, you must specify an array of formats
72  * and an array of limits. The length of these arrays must be the same.
73  * For example,
74  * <ul>
75  * <li>
76  * <em>limits = {1,2,3,4,5,6,7}<br>
77  * <em>formats = {"Sun", "Mon", "Tue", "Wed", "Thur", "Fri", "Sat"}
78  * </li>
79  * <em>limits = {0, 1, ChoiceFormat.nextDouble(1)}<br>
80  * <em>formats = {"no files", "one file", "many files"}<br>
81  * (<code>nextDouble can be used to get the next higher double, to
82  * make the half-open interval.)
83  * </ul>
84  *
85  * <p>
86  * Here is a simple example that shows formatting and parsing:
87  * <blockquote>
88  * <pre>{@code
89  * double[] limits = {1,2,3,4,5,6,7};
90  * String[] dayOfWeekNames = {"Sun", "Mon", "Tue", "Wed", "Thur", "Fri", "Sat"};
91  * ChoiceFormat form = new ChoiceFormat(limits, dayOfWeekNames);
92  * ParsePosition status = new ParsePosition(0);
93  * for (double i = 0.0; i <= 8.0; ++i) {
94  *     status.setIndex(0);
95  *     System.out.println(i + " -> " + form.format(i) + " -> "

```

```

96         *             + form.parse(form.format(i),status));
97     * }
98     * }</pre>
99     * </blockquote>
100    * Here is a more complex example, with a pattern format:
101    * <blockquote>
102    * <pre>{@code
103    * double[] filelimits = {0,1,2};
104    * String[] filepart = {"are no files","is one file","are {2} files"};
105    * ChoiceFormat fileform = new ChoiceFormat(filelimits, filepart);
106    * Format[] testFormats = {fileform, null, NumberFormat.getInstance()};
107    * MessageFormat pattform = new MessageFormat("There {0} on {1}");
108    * pattform.setFormats(testFormats);
109    * Object[] testArgs = {null, "ADisk", null};
110    * for (int i = 0; i < 4; ++i) {
111    *     testArgs[0] = new Integer(i);
112    *     testArgs[2] = testArgs[0];
113    *     System.out.println(pattform.format(testArgs));
114    * }
115    * }</pre>
116    * </blockquote>
117    * <p>
118    * Specifying a pattern for ChoiceFormat objects is fairly straightforward.
119    * For example:
120    * <blockquote>
121    * <pre>{@code
122    * ChoiceFormat fmt = new ChoiceFormat(
123    *     "-1#is negative| 0#is zero or fraction | 1#is one |1.0<is 1+ |2#is two |2<is more than 2.")
124    *     ;
125    * System.out.println("Formatter Pattern : " + fmt.toPattern());
126    *
127    * System.out.println("Format with -INF : " + fmt.format(Double.NEGATIVE_INFINITY));
128    * System.out.println("Format with -1.0 : " + fmt.format(-1.0));
129    * System.out.println("Format with 0 : " + fmt.format(0));
130    * System.out.println("Format with 0.9 : " + fmt.format(0.9));
131    * System.out.println("Format with 1.0 : " + fmt.format(1));
132    * System.out.println("Format with 1.5 : " + fmt.format(1.5));
133    * System.out.println("Format with 2 : " + fmt.format(2));
134    * System.out.println("Format with 2.1 : " + fmt.format(2.1));
135    * System.out.println("Format with NaN : " + fmt.format(Double.NaN));
136    * System.out.println("Format with +INF : " + fmt.format(Double.POSITIVE_INFINITY));
137    * }</pre>
138    * </blockquote>
139    * And the output result would be like the following:
140    * <blockquote>
141    * <pre>{@code
142    * Format with -INF : is negative
143    * Format with -1.0 : is negative
144    * Format with 0 : is zero or fraction
145    * Format with 0.9 : is zero or fraction
146    * Format with 1.0 : is one
147    * Format with 1.5 : is 1+
148    * Format with 2 : is two

```

```

148 * Format with 2.1 : is more than 2.
149 * Format with NaN : is negative
150 * Format with +INF : is more than 2.
151 * }</pre>
152 * </blockquote>
153 *
154 * <h3><a name="synchronization">Synchronization</a></h3>
155 *
156 * <p>
157 * Choice formats are not synchronized.
158 * It is recommended to create separate format instances for each thread.
159 * If multiple threads access a format concurrently, it must be synchronized
160 * externally.
161 *
162 *
163 * @see      DecimalFormat
164 * @see      MessageFormat
165 * @author    Mark Davis
166 */
167 public class ChoiceFormat extends NumberFormat {
168
169     // Proclaim serial compatibility with 1.1 FCS
170     private static final long serialVersionUID = 1795184449645032964L;
171
172     /**
173      * Sets the pattern.
174      * @param newPattern See the class description.
175      */
176     public void applyPattern(String newPattern) {
177         StringBuffer[] segments = new StringBuffer[2];
178         for (int i = 0; i < segments.length; ++i) {
179             segments[i] = new StringBuffer();
180         }
181         double[] newChoiceLimits = new double[30];
182         String[] newChoiceFormats = new String[30];
183         int count = 0;
184         int part = 0;
185         double startValue = 0;
186         double oldStartValue = Double.NaN;
187         boolean inQuote = false;
188         for (int i = 0; i < newPattern.length(); ++i) {
189             char ch = newPattern.charAt(i);
190             if (ch=='\\') {
191                 // Check for "" indicating a literal quote
192                 if ((i+1)<newPattern.length() && newPattern.charAt(i+1)==ch) {
193                     segments[part].append(ch);
194                     ++i;
195                 } else {
196                     inQuote = !inQuote;
197                 }
198             } else if (inQuote) {
199                 segments[part].append(ch);
200             } else if (ch == '<' || ch == '#' || ch == '\u2264') {

```



```

201         if (segments[0].length() == 0) {
202             throw new IllegalArgumentException();
203         }
204         try {
205             String tempBuffer = segments[0].toString();
206             if (tempBuffer.equals("\u221E")) {
207                 startValue = Double.POSITIVE_INFINITY;
208             } else if (tempBuffer.equals("-\u221E")) {
209                 startValue = Double.NEGATIVE_INFINITY;
210             } else {
211                 startValue = Double.valueOf(segments[0].toString()).doubleValue();
212             }
213         } catch (Exception e) {
214             throw new IllegalArgumentException();
215         }
216         if (ch == '<' && startValue != Double.POSITIVE_INFINITY &&
217             startValue != Double.NEGATIVE_INFINITY) {
218             startValue = nextDouble(startValue);
219         }
220         if (startValue <= oldStartValue) {
221             throw new IllegalArgumentException();
222         }
223         segments[0].setLength(0);
224         part = 1;
225     } else if (ch == '|') {
226         if (count == newChoiceLimits.length) {
227             newChoiceLimits = doubleArraySize(newChoiceLimits);
228             newChoiceFormats = doubleArraySize(newChoiceFormats);
229         }
230         newChoiceLimits[count] = startValue;
231         newChoiceFormats[count] = segments[1].toString();
232         ++count;
233         oldStartValue = startValue;
234         segments[1].setLength(0);
235         part = 0;
236     } else {
237         segments[part].append(ch);
238     }
239 }
240 // clean up last one
241 if (part == 1) {
242     if (count == newChoiceLimits.length) {
243         newChoiceLimits = doubleArraySize(newChoiceLimits);
244         newChoiceFormats = doubleArraySize(newChoiceFormats);
245     }
246     newChoiceLimits[count] = startValue;
247     newChoiceFormats[count] = segments[1].toString();
248     ++count;
249 }
250 choiceLimits = new double[count];
251 System.arraycopy(newChoiceLimits, 0, choiceLimits, 0, count);
252 choiceFormats = new String[count];
253 System.arraycopy(newChoiceFormats, 0, choiceFormats, 0, count);

```

```

254     }
255
256     /**
257      * Gets the pattern.
258      *
259      * @return the pattern string
260      */
261     public String toPattern() {
262         StringBuffer result = new StringBuffer();
263         for (int i = 0; i < choiceLimits.length; ++i) {
264             if (i != 0) {
265                 result.append('|');
266             }
267             // choose based upon which has less precision
268             // approximate that by choosing the closest one to an integer.
269             // could do better, but it's not worth it.
270             double less = previousDouble(choiceLimits[i]);
271             double tryLessOrEqual = Math.abs(Math.IEEEremainder(choiceLimits[i], 1.0d));
272             double tryLess = Math.abs(Math.IEEEremainder(less, 1.0d));
273
274             if (tryLessOrEqual < tryLess) {
275                 result.append(""+choiceLimits[i]);
276                 result.append('#');
277             } else {
278                 if (choiceLimits[i] == Double.POSITIVE_INFINITY) {
279                     result.append("\u221E");
280                 } else if (choiceLimits[i] == Double.NEGATIVE_INFINITY) {
281                     result.append("-\u221E");
282                 } else {
283                     result.append(""+less);
284                 }
285                 result.append('<');
286             }
287             // Append choiceFormats[i], using quotes if there are special characters.
288             // Single quotes themselves must be escaped in either case.
289             String text = choiceFormats[i];
290             boolean needQuote = text.indexOf('<') >= 0
291                 || text.indexOf('#') >= 0
292                 || text.indexOf('\u2264') >= 0
293                 || text.indexOf('|') >= 0;
294             if (needQuote) result.append('\\');
295             if (text.indexOf('\\') < 0) result.append(text);
296             else {
297                 for (int j=0; j<text.length(); ++j) {
298                     char c = text.charAt(j);
299                     result.append(c);
300                     if (c == '\\') result.append(c);
301                 }
302             }
303             if (needQuote) result.append('\\');
304         }
305         return result.toString();
306     }

```

```

307
308 /**
309  * Constructs with limits and corresponding formats based on the pattern.
310  *
311  * @param newPattern the new pattern string
312  * @see #applyPattern
313  */
314 public ChoiceFormat(String newPattern) {
315     applyPattern(newPattern);
316 }
317
318 /**
319  * Constructs with the limits and the corresponding formats.
320  *
321  * @param limits limits in ascending order
322  * @param formats corresponding format strings
323  * @see #setChoices
324  */
325 public ChoiceFormat(double[] limits, String[] formats) {
326     setChoices(limits, formats);
327 }
328
329 /**
330  * Set the choices to be used in formatting.
331  * @param limits contains the top value that you want
332  *   parsed with that format, and should be in ascending sorted order. When
333  *   formatting X, the choice will be the i, where
334  *   limit[i] ≤ X < limit[i+1].
335  * If the limit array is not in ascending order, the results of formatting
336  * will be incorrect.
337  * @param formats are the formats you want to use for each limit.
338  * They can be either Format objects or Strings.
339  * When formatting with object Y,
340  * if the object is a NumberFormat, then ((NumberFormat) Y).format(X)
341  * is called. Otherwise Y.toString() is called.
342  */
343 public void setChoices(double[] limits, String formats[]) {
344     if (limits.length != formats.length) {
345         throw new IllegalArgumentException(
346             "Array and limit arrays must be of the same length.");
347     }
348     choiceLimits = Arrays.copyOf(limits, limits.length);
349     choiceFormats = Arrays.copyOf(formats, formats.length);
350 }
351
352 /**
353  * Get the limits passed in the constructor.
354  * @return the limits.
355  */
356 public double[] getLimits() {
357     double[] newLimits = Arrays.copyOf(choiceLimits, choiceLimits.length);
358     return newLimits;
359 }

```

```

360
361 /**
362  * Get the formats passed in the constructor.
363  * @return the formats.
364  */
365 public Object[] getFormats() {
366     Object[] newFormats = Arrays.copyOf(choiceFormats, choiceFormats.length);
367     return newFormats;
368 }
369
370 // Overrides
371
372 /**
373  * Specialization of format. This method really calls
374  * <code>format(double, StringBuffer, FieldPosition)</code>
375  * thus the range of longs that are supported is only equal to
376  * the range that can be stored by double. This will never be
377  * a practical limitation.
378  */
379 public StringBuffer format(long number, StringBuffer toAppendTo,
380                             FieldPosition status) {
381     return format((double)number, toAppendTo, status);
382 }
383
384 /**
385  * Returns pattern with formatted double.
386  * @param number number to be formatted and substituted.
387  * @param toAppendTo where text is appended.
388  * @param status ignore no useful status is returned.
389  */
390 public StringBuffer format(double number, StringBuffer toAppendTo,
391                             FieldPosition status) {
392     // find the number
393     int i;
394     for (i = 0; i < choiceLimits.length; ++i) {
395         if (!(number >= choiceLimits[i])) {
396             // same as number < choiceLimits, except catches NaN
397             break;
398         }
399     }
400     --i;
401     if (i < 0) i = 0;
402     // return either a formatted number, or a string
403     return toAppendTo.append(choiceFormats[i]);
404 }
405
406 /**
407  * Parses a Number from the input text.
408  * @param text the source text.
409  * @param status an input-output parameter. On input, the
410  * status.index field indicates the first character of the
411  * source text that should be parsed. On exit, if no error
412  * occurred, status.index is set to the first unparsed character

```

```

413     * in the source text. On exit, if an error did occur,
414     * status.index is unchanged and status.errorIndex is set to the
415     * first index of the character that caused the parse to fail.
416     * @return A Number representing the value of the number parsed.
417     */
418     public Number parse(String text, ParsePosition status) {
419         // find the best number (defined as the one with the longest parse)
420         int start = status.index;
421         int furthest = start;
422         double bestNumber = Double.NaN;
423         double tempNumber = 0.0;
424         for (int i = 0; i < choiceFormats.length; ++i) {
425             String tempString = choiceFormats[i];
426             if (text.regionMatches(start, tempString, 0, tempString.length())) {
427                 status.index = start + tempString.length();
428                 tempNumber = choiceLimits[i];
429                 if (status.index > furthest) {
430                     furthest = status.index;
431                     bestNumber = tempNumber;
432                     if (furthest == text.length()) break;
433                 }
434             }
435         }
436         status.index = furthest;
437         if (status.index == start) {
438             status.errorIndex = furthest;
439         }
440         return new Double(bestNumber);
441     }
442
443     /**
444     * Finds the least double greater than {@code d}.
445     * If {@code NaN}, returns same value.
446     * <p>Used to make half-open intervals.
447     *
448     * @param d the reference value
449     * @return the least double value greater than {@code d}
450     * @see #previousDouble
451     */
452     public static final double nextDouble (double d) {
453         return nextDouble(d,true);
454     }
455
456     /**
457     * Finds the greatest double less than {@code d}.
458     * If {@code NaN}, returns same value.
459     *
460     * @param d the reference value
461     * @return the greatest double value less than {@code d}
462     * @see #nextDouble
463     */
464     public static final double previousDouble (double d) {
465         return nextDouble(d,false);

```

```
466     }
467
468     /**
469     * Overrides Cloneable
470     */
471     public Object clone()
472     {
473         ChoiceFormat other = (ChoiceFormat) super.clone();
474         // for primitives or immutables, shallow clone is enough
475         other.choiceLimits = choiceLimits.clone();
476         other.choiceFormats = choiceFormats.clone();
477         return other;
478     }
479
480     /**
481     * Generates a hash code for the message format object.
482     */
483     public int hashCode() {
484         int result = choiceLimits.length;
485         if (choiceFormats.length > 0) {
486             // enough for reasonable distribution
487             result ^= choiceFormats[choiceFormats.length-1].hashCode();
488         }
489         return result;
490     }
491
492     /**
493     * Equality comparison between two
494     */
495     public boolean equals(Object obj) {
496         if (obj == null) return false;
497         if (this == obj)                // quick check
498             return true;
499         if (getClass() != obj.getClass())
500             return false;
501         ChoiceFormat other = (ChoiceFormat) obj;
502         return (Arrays.equals(choiceLimits, other.choiceLimits)
503             && Arrays.equals(choiceFormats, other.choiceFormats));
504     }
505
506     /**
507     * After reading an object from the input stream, do a simple verification
508     * to maintain class invariants.
509     * @throws InvalidObjectException if the objects read from the stream is invalid.
510     */
511     private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
512         in.defaultReadObject();
513         if (choiceLimits.length != choiceFormats.length) {
514             throw new InvalidObjectException(
515                 "limits and format arrays of different length.");
516         }
517     }
518 }
```

```

519 // =====privates=====
520
521 /**
522  * A list of lower bounds for the choices. The formatter will return
523  * <code>choiceFormats[i]</code> if the number being formatted is greater than or equal to
524  * <code>choiceLimits[i]</code> and less than <code>choiceLimits[i+1]</code>.
525  * @serial
526  */
527 private double[] choiceLimits;
528
529 /**
530  * A list of choice strings. The formatter will return
531  * <code>choiceFormats[i]</code> if the number being formatted is greater than or equal to
532  * <code>choiceLimits[i]</code> and less than <code>choiceLimits[i+1]</code>.
533  * @serial
534  */
535 private String[] choiceFormats;
536
537 /*
538  static final long SIGN          = 0x8000000000000000L;
539  static final long EXPONENT      = 0x7FF0000000000000L;
540  static final long SIGNIFICAND   = 0x00FFFFFFFFFFFFFL;
541
542  private static double nextDouble (double d, boolean positive) {
543      if (Double.isNaN(d) || Double.isInfinite(d)) {
544          return d;
545      }
546      long bits = Double.doubleToLongBits(d);
547      long significand = bits & SIGNIFICAND;
548      if (bits < 0) {
549          significand |= (SIGN | EXPONENT);
550      }
551      long exponent = bits & EXPONENT;
552      if (positive) {
553          significand += 1;
554          // FIXME fix overflow & underflow
555      } else {
556          significand -= 1;
557          // FIXME fix overflow & underflow
558      }
559      bits = exponent | (significand & ~EXPONENT);
560      return Double.longBitsToDouble(bits);
561  }
562  */
563
564  static final long SIGN          = 0x8000000000000000L;
565  static final long EXPONENT      = 0x7FF0000000000000L;
566  static final long POSITIVEINFINITY = 0x7FF0000000000000L;
567
568  /**
569   * Finds the least double greater than {@code d} (if {@code positive} is
570   * {@code true}), or the greatest double less than {@code d} (if
571   * {@code positive} is {@code false}).

```

```

572     * If {@code NaN}, returns same value.
573     *
574     * Does not affect floating-point flags,
575     * provided these member functions do not:
576     *     Double.longBitsToDouble(long)
577     *     Double.doubleToLongBits(double)
578     *     Double.isNaN(double)
579     *
580     * @param d        the reference value
581     * @param positive {@code true} if the least double is desired;
582     *                {@code false} otherwise
583     * @return the least or greater double value
584     */
585     public static double nextDouble (double d, boolean positive) {
586
587         /* filter out NaN's */
588         if (Double.isNaN(d)) {
589             return d;
590         }
591
592         /* zero's are also a special case */
593         if (d == 0.0) {
594             double smallestPositiveDouble = Double.longBitsToDouble(1L);
595             if (positive) {
596                 return smallestPositiveDouble;
597             } else {
598                 return -smallestPositiveDouble;
599             }
600         }
601
602         /* if entering here, d is a nonzero value */
603
604         /* hold all bits in a long for later use */
605         long bits = Double.doubleToLongBits(d);
606
607         /* strip off the sign bit */
608         long magnitude = bits & ~SIGN;
609
610         /* if next double away from zero, increase magnitude */
611         if ((bits > 0) == positive) {
612             if (magnitude != POSITIVEINFINITY) {
613                 magnitude += 1;
614             }
615         }
616         /* else decrease magnitude */
617         else {
618             magnitude -= 1;
619         }
620
621         /* restore sign bit and return */
622         long signbit = bits & SIGN;
623         return Double.longBitsToDouble (magnitude | signbit);
624     }

```



```

625
626     private static double[] doubleArraySize(double[] array) {
627         int oldSize = array.length;
628         double[] newArray = new double[oldSize * 2];
629         System.arraycopy(array, 0, newArray, 0, oldSize);
630         return newArray;
631     }
632
633     private String[] doubleArraySize(String[] array) {
634         int oldSize = array.length;
635         String[] newArray = new String[oldSize * 2];
636         System.arraycopy(array, 0, newArray, 0, oldSize);
637         return newArray;
638     }
639
640 }

```

2.10 CollationElementIterator.java

Secuencia 19: java/text/CollationElementIterator.java

```

1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28 * (C) Copyright IBM Corp. 1996-1998 - All Rights Reserved
29 *
30 * The original version of this source code and documentation is copyrighted
31 * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32 * materials are provided under terms of a License Agreement between Taligent
33 * and Sun. This technology is protected by multiple US and International

```

```
34 * patents. This notice and attribution to Taligent may not be removed.
35 * Taligent is a registered trademark of Taligent, Inc.
36 *
37 */
38
39 package java.text;
40
41 import java.lang.Character;
42 import java.util.Vector;
43 import sun.text.CollatorUtilities;
44 import sun.text.normalizer.NormalizerBase;
45
46 /**
47 * The <code>CollationElementIterator</code> class is used as an iterator
48 * to walk through each character of an international string. Use the iterator
49 * to return the ordering priority of the positioned character. The ordering
50 * priority of a character, which we refer to as a key, defines how a character
51 * is collated in the given collation object.
52 *
53 * <p>
54 * For example, consider the following in Spanish:
55 * <blockquote>
56 * <pre>
57 * "ca" &rarr; the first key is key('c') and second key is key('a').
58 * "cha" &rarr; the first key is key('ch') and second key is key('a').
59 * </pre>
60 * </blockquote>
61 * And in German,
62 * <blockquote>
63 * <pre>
64 * "\u00e4b" &rarr; the first key is key('a'), the second key is key('e'), and
65 * the third key is key('b').
66 * </pre>
67 * </blockquote>
68 * The key of a character is an integer composed of primary order(short),
69 * secondary order(byte), and tertiary order(byte). Java strictly defines
70 * the size and signedness of its primitive data types. Therefore, the static
71 * functions <code>primaryOrder</code>, <code>secondaryOrder</code>, and
72 * <code>tertiaryOrder</code> return <code>int</code>, <code>short</code>,
73 * and <code>short</code> respectively to ensure the correctness of the key
74 * value.
75 *
76 * <p>
77 * Example of the iterator usage,
78 * <blockquote>
79 * <pre>
80 *
81 * String testString = "This is a test";
82 * Collator col = Collator.getInstance();
83 * if (col instanceof RuleBasedCollator) {
84 *     RuleBasedCollator ruleBasedCollator = (RuleBasedCollator)col;
85 *     CollationElementIterator collationElementIterator = ruleBasedCollator.
86 *         getCollationElementIterator(testString);
```

```

86 *      int primaryOrder = CollationElementIterator.primaryOrder(collationElementIterator.next());
87 *      :
88 *  }
89 * </pre>
90 * </blockquote>
91 *
92 * <p>
93 * <code>CollationElementIterator.next</code> returns the collation order
94 * of the next character. A collation order consists of primary order,
95 * secondary order and tertiary order. The data type of the collation
96 * order is <strong>int</strong>. The first 16 bits of a collation order
97 * is its primary order; the next 8 bits is the secondary order and the
98 * last 8 bits is the tertiary order.
99 *
100 * <p><b>Note:</b> <code>CollationElementIterator</code> is a part of
101 * <code>RuleBasedCollator</code> implementation. It is only usable
102 * with <code>RuleBasedCollator</code> instances.
103 *
104 * @see          Collator
105 * @see          RuleBasedCollator
106 * @author       Helena Shih, Laura Werner, Richard Gillam
107 */
108 public final class CollationElementIterator
109 {
110     /**
111      * Null order which indicates the end of string is reached by the
112      * cursor.
113      */
114     public final static int NULLORDER = 0xffffffff;
115
116     /**
117      * CollationElementIterator constructor. This takes the source string and
118      * the collation object. The cursor will walk thru the source string based
119      * on the predefined collation rules. If the source string is empty,
120      * NULLORDER will be returned on the calls to next().
121      * @param sourceText the source string.
122      * @param owner the collation object.
123      */
124     CollationElementIterator(String sourceText, RuleBasedCollator owner) {
125         this.owner = owner;
126         ordering = owner.getTables();
127         if ( sourceText.length() != 0 ) {
128             NormalizerBase.Mode mode =
129                 CollatorUtilities.toNormalizerMode(owner.getDecomposition());
130             text = new NormalizerBase(sourceText, mode);
131         }
132     }
133
134     /**
135      * CollationElementIterator constructor. This takes the source string and
136      * the collation object. The cursor will walk thru the source string based
137      * on the predefined collation rules. If the source string is empty,
138      * NULLORDER will be returned on the calls to next().

```

```

139     * @param sourceText the source string.
140     * @param owner the collation object.
141     */
142     CollationElementIterator(CharacterIterator sourceText, RuleBasedCollator owner) {
143         this.owner = owner;
144         ordering = owner.getTables();
145         NormalizerBase.Mode mode =
146             CollatorUtilities.toNormalizerMode(owner.getDecomposition());
147         text = new NormalizerBase(sourceText, mode);
148     }
149
150     /**
151     * Resets the cursor to the beginning of the string. The next call
152     * to next() will return the first collation element in the string.
153     */
154     public void reset()
155     {
156         if (text != null) {
157             text.reset();
158             NormalizerBase.Mode mode =
159                 CollatorUtilities.toNormalizerMode(owner.getDecomposition());
160             text.setMode(mode);
161         }
162         buffer = null;
163         expIndex = 0;
164         swapOrder = 0;
165     }
166
167     /**
168     * Get the next collation element in the string. <p>This iterator iterates
169     * over a sequence of collation elements that were built from the string.
170     * Because there isn't necessarily a one-to-one mapping from characters to
171     * collation elements, this doesn't mean the same thing as "return the
172     * collation element [or ordering priority] of the next character in the
173     * string".</p>
174     * <p>This function returns the collation element that the iterator is currently
175     * pointing to and then updates the internal pointer to point to the next element.
176     * previous() updates the pointer first and then returns the element. This
177     * means that when you change direction while iterating (i.e., call next() and
178     * then call previous(), or call previous() and then call next()), you'll get
179     * back the same element twice.</p>
180     *
181     * @return the next collation element
182     */
183     public int next()
184     {
185         if (text == null) {
186             return NULLORDER;
187         }
188         NormalizerBase.Mode textMode = text.getMode();
189         // convert the owner's mode to something the Normalizer understands
190         NormalizerBase.Mode ownerMode =
191             CollatorUtilities.toNormalizerMode(owner.getDecomposition());

```

```

192     if (textMode != ownerMode) {
193         text.setMode(ownerMode);
194     }
195
196     // if buffer contains any decomposed char values
197     // return their strength orders before continuing in
198     // the Normalizer's CharacterIterator.
199     if (buffer != null) {
200         if (expIndex < buffer.length) {
201             return strengthOrder(buffer[expIndex++]);
202         } else {
203             buffer = null;
204             expIndex = 0;
205         }
206     } else if (swapOrder != 0) {
207         if (Character.isSupplementaryCodePoint(swapOrder)) {
208             char[] chars = Character.toChars(swapOrder);
209             swapOrder = chars[1];
210             return chars[0] << 16;
211         }
212         int order = swapOrder << 16;
213         swapOrder = 0;
214         return order;
215     }
216     int ch = text.next();
217
218     // are we at the end of Normalizer's text?
219     if (ch == NormalizerBase.DONE) {
220         return NULLORDER;
221     }
222
223     int value = ordering.getUnicodeOrder(ch);
224     if (value == RuleBasedCollator.UNMAPPED) {
225         swapOrder = ch;
226         return UNMAPPEDCHARVALUE;
227     }
228     else if (value >= RuleBasedCollator.CONTRACTCHARINDEX) {
229         value = nextContractChar(ch);
230     }
231     if (value >= RuleBasedCollator.EXPANDCHARINDEX) {
232         buffer = ordering.getExpandValueList(value);
233         expIndex = 0;
234         value = buffer[expIndex++];
235     }
236
237     if (ordering.isSEAsianSwapping()) {
238         int consonant;
239         if (isThaiPreVowel(ch)) {
240             consonant = text.next();
241             if (isThaiBaseConsonant(consonant)) {
242                 buffer = makeReorderedBuffer(consonant, value, buffer, true);
243                 value = buffer[0];
244                 expIndex = 1;

```

```

245         } else if (consonant != NormalizerBase.DONE) {
246             text.previous();
247         }
248     }
249     if (isLaoPreVowel(ch)) {
250         consonant = text.next();
251         if (isLaoBaseConsonant(consonant)) {
252             buffer = makeReorderedBuffer(consonant, value, buffer, true);
253             value = buffer[0];
254             expIndex = 1;
255         } else if (consonant != NormalizerBase.DONE) {
256             text.previous();
257         }
258     }
259 }
260
261     return strengthOrder(value);
262 }
263
264 /**
265  * Get the previous collation element in the string. <p>This iterator iterates
266  * over a sequence of collation elements that were built from the string.
267  * Because there isn't necessarily a one-to-one mapping from characters to
268  * collation elements, this doesn't mean the same thing as "return the
269  * collation element [or ordering priority] of the previous character in the
270  * string".</p>
271  * <p>This function updates the iterator's internal pointer to point to the
272  * collation element preceding the one it's currently pointing to and then
273  * returns that element, while next() returns the current element and then
274  * updates the pointer. This means that when you change direction while
275  * iterating (i.e., call next() and then call previous(), or call previous()
276  * and then call next()), you'll get back the same element twice.</p>
277  *
278  * @return the previous collation element
279  * @since 1.2
280  */
281 public int previous()
282 {
283     if (text == null) {
284         return NULLORDER;
285     }
286     NormalizerBase.Mode textMode = text.getMode();
287     // convert the owner's mode to something the Normalizer understands
288     NormalizerBase.Mode ownerMode =
289         CollatorUtilities.toNormalizerMode(owner.getDecomposition());
290     if (textMode != ownerMode) {
291         text.setMode(ownerMode);
292     }
293     if (buffer != null) {
294         if (expIndex > 0) {
295             return strengthOrder(buffer[--expIndex]);
296         } else {
297             buffer = null;

```

```

298         expIndex = 0;
299     }
300 } else if (swapOrder != 0) {
301     if (Character.isSupplementaryCodePoint(swapOrder)) {
302         char[] chars = Character.toChars(swapOrder);
303         swapOrder = chars[1];
304         return chars[0] << 16;
305     }
306     int order = swapOrder << 16;
307     swapOrder = 0;
308     return order;
309 }
310 int ch = text.previous();
311 if (ch == NormalizerBase.DONE) {
312     return NULLORDER;
313 }
314
315 int value = ordering.getUnicodeOrder(ch);
316
317 if (value == RuleBasedCollator.UNMAPPED) {
318     swapOrder = UNMAPPEDCHARVALUE;
319     return ch;
320 } else if (value >= RuleBasedCollator.CONTRACTCHARINDEX) {
321     value = prevContractChar(ch);
322 }
323 if (value >= RuleBasedCollator.EXPANDCHARINDEX) {
324     buffer = ordering.getExpandValueList(value);
325     expIndex = buffer.length;
326     value = buffer[--expIndex];
327 }
328
329 if (ordering.isSEAsianSwapping()) {
330     int vowel;
331     if (isThaiBaseConsonant(ch)) {
332         vowel = text.previous();
333         if (isThaiPreVowel(vowel)) {
334             buffer = makeReorderedBuffer(vowel, value, buffer, false);
335             expIndex = buffer.length - 1;
336             value = buffer[expIndex];
337         } else {
338             text.next();
339         }
340     }
341     if (isLaoBaseConsonant(ch)) {
342         vowel = text.previous();
343         if (isLaoPreVowel(vowel)) {
344             buffer = makeReorderedBuffer(vowel, value, buffer, false);
345             expIndex = buffer.length - 1;
346             value = buffer[expIndex];
347         } else {
348             text.next();
349         }
350     }

```

```

351     }
352
353     return strengthOrder(value);
354 }
355
356 /**
357  * Return the primary component of a collation element.
358  * @param order the collation element
359  * @return the element's primary component
360  */
361 public final static int primaryOrder(int order)
362 {
363     order &= RBCollationTables.PRIMARYORDERMASK;
364     return (order >>> RBCollationTables.PRIMARYORDERSHIFT);
365 }
366 /**
367  * Return the secondary component of a collation element.
368  * @param order the collation element
369  * @return the element's secondary component
370  */
371 public final static short secondaryOrder(int order)
372 {
373     order = order & RBCollationTables.SECONDARYORDERMASK;
374     return ((short)(order >> RBCollationTables.SECONDARYORDERSHIFT));
375 }
376 /**
377  * Return the tertiary component of a collation element.
378  * @param order the collation element
379  * @return the element's tertiary component
380  */
381 public final static short tertiaryOrder(int order)
382 {
383     return ((short)(order &= RBCollationTables.TERTIARYORDERMASK));
384 }
385
386 /**
387  * Get the comparison order in the desired strength. Ignore the other
388  * differences.
389  * @param order The order value
390  */
391 final int strengthOrder(int order)
392 {
393     int s = owner.getStrength();
394     if (s == Collator.PRIMARY)
395     {
396         order &= RBCollationTables.PRIMARYDIFFERENCEONLY;
397     } else if (s == Collator.SECONDARY)
398     {
399         order &= RBCollationTables.SECONDARYDIFFERENCEONLY;
400     }
401     return order;
402 }
403

```



```

404  /**
405   * Sets the iterator to point to the collation element corresponding to
406   * the specified character (the parameter is a CHARACTER offset in the
407   * original string, not an offset into its corresponding sequence of
408   * collation elements). The value returned by the next call to next()
409   * will be the collation element corresponding to the specified position
410   * in the text. If that position is in the middle of a contracting
411   * character sequence, the result of the next call to next() is the
412   * collation element for that sequence. This means that getOffset()
413   * is not guaranteed to return the same value as was passed to a preceding
414   * call to setOffset().
415   *
416   * @param newOffset The new character offset into the original text.
417   * @since 1.2
418   */
419  @SuppressWarnings("deprecation") // getBeginIndex, getEndIndex and setIndex are deprecated
420  public void setOffset(int newOffset)
421  {
422      if (text != null) {
423          if (newOffset < text.getBeginIndex()
424              || newOffset >= text.getEndIndex()) {
425              text.setIndexOnly(newOffset);
426          } else {
427              int c = text.setIndex(newOffset);
428
429              // if the desired character isn't used in a contracting character
430              // sequence, bypass all the backing-up logic-- we're sitting on
431              // the right character already
432              if (ordering.usedInContractSeq(c)) {
433                  // walk backwards through the string until we see a character
434                  // that DOESN'T participate in a contracting character sequence
435                  while (ordering.usedInContractSeq(c)) {
436                      c = text.previous();
437                  }
438                  // now walk forward using this object's next() method until
439                  // we pass the starting point and set our current position
440                  // to the beginning of the last "character" before or at
441                  // our starting position
442                  int last = text.getIndex();
443                  while (text.getIndex() <= newOffset) {
444                      last = text.getIndex();
445                      next();
446                  }
447                  text.setIndexOnly(last);
448                  // we don't need this, since last is the last index
449                  // that is the starting of the contraction which encompass
450                  // newOffset
451                  // text.previous();
452              }
453          }
454      }
455      buffer = null;
456      expIndex = 0;

```

```

457         swapOrder = 0;
458     }
459
460     /**
461      * Returns the character offset in the original text corresponding to the next
462      * collation element. (That is, getOffset() returns the position in the text
463      * corresponding to the collation element that will be returned by the next
464      * call to next().) This value will always be the index of the FIRST character
465      * corresponding to the collation element (a contracting character sequence is
466      * when two or more characters all correspond to the same collation element).
467      * This means if you do setOffset(x) followed immediately by getOffset(), getOffset()
468      * won't necessarily return x.
469      *
470      * @return The character offset in the original text corresponding to the collation
471      * element that will be returned by the next call to next().
472      * @since 1.2
473      */
474     public int getOffset()
475     {
476         return (text != null) ? text.getIndex() : 0;
477     }
478
479
480     /**
481      * Return the maximum length of any expansion sequences that end
482      * with the specified comparison order.
483      * @param order a collation order returned by previous or next.
484      * @return the maximum length of any expansion sequences ending
485      *         with the specified order.
486      * @since 1.2
487      */
488     public int getMaxExpansion(int order)
489     {
490         return ordering.getMaxExpansion(order);
491     }
492
493     /**
494      * Set a new string over which to iterate.
495      *
496      * @param source the new source text
497      * @since 1.2
498      */
499     public void setText(String source)
500     {
501         buffer = null;
502         swapOrder = 0;
503         expIndex = 0;
504         NormalizerBase.Mode mode =
505             CollatorUtilities.toNormalizerMode(owner.getDecomposition());
506         if (text == null) {
507             text = new NormalizerBase(source, mode);
508         } else {
509             text.setMode(mode);

```

```

510         text.setText(source);
511     }
512 }
513
514 /**
515  * Set a new string over which to iterate.
516  *
517  * @param source the new source text.
518  * @since 1.2
519  */
520 public void setText(CharacterIterator source)
521 {
522     buffer = null;
523     swapOrder = 0;
524     expIndex = 0;
525     NormalizerBase.Mode mode =
526         CollatorUtilities.toNormalizerMode(owner.getDecomposition());
527     if (text == null) {
528         text = new NormalizerBase(source, mode);
529     } else {
530         text.setMode(mode);
531         text.setText(source);
532     }
533 }
534
535 //=====
536 // privates
537 //=====
538
539 /**
540  * Determine if a character is a Thai vowel (which sorts after
541  * its base consonant).
542  */
543 private final static boolean isThaiPreVowel(int ch) {
544     return (ch >= 0x0e40) && (ch <= 0x0e44);
545 }
546
547 /**
548  * Determine if a character is a Thai base consonant
549  */
550 private final static boolean isThaiBaseConsonant(int ch) {
551     return (ch >= 0x0e01) && (ch <= 0x0e2e);
552 }
553
554 /**
555  * Determine if a character is a Lao vowel (which sorts after
556  * its base consonant).
557  */
558 private final static boolean isLaoPreVowel(int ch) {
559     return (ch >= 0x0ec0) && (ch <= 0x0ec4);
560 }
561
562 /**

```

```

563     * Determine if a character is a Lao base consonant
564     */
565     private final static boolean isLaoBaseConsonant(int ch) {
566         return (ch >= 0x0e81) && (ch <= 0x0eae);
567     }
568
569     /**
570     * This method produces a buffer which contains the collation
571     * elements for the two characters, with colFirst's values preceding
572     * another character's. Presumably, the other character precedes colFirst
573     * in logical order (otherwise you wouldn't need this method would you?).
574     * The assumption is that the other char's value(s) have already been
575     * computed. If this char has a single element it is passed to this
576     * method as lastValue, and lastExpansion is null. If it has an
577     * expansion it is passed in lastExpansion, and collLastValue is ignored.
578     */
579     private int[] makeReorderedBuffer(int colFirst,
580                                     int lastValue,
581                                     int[] lastExpansion,
582                                     boolean forward) {
583
584         int[] result;
585
586         int firstValue = ordering.getUnicodeOrder(colFirst);
587         if (firstValue >= RuleBasedCollator.CONTRACTCHARINDEX) {
588             firstValue = forward? nextContractChar(colFirst) : prevContractChar(colFirst);
589         }
590
591         int[] firstExpansion = null;
592         if (firstValue >= RuleBasedCollator.EXPANDCHARINDEX) {
593             firstExpansion = ordering.getExpandValueList(firstValue);
594         }
595
596         if (!forward) {
597             int temp1 = firstValue;
598             firstValue = lastValue;
599             lastValue = temp1;
600             int[] temp2 = firstExpansion;
601             firstExpansion = lastExpansion;
602             lastExpansion = temp2;
603         }
604
605         if (firstExpansion == null && lastExpansion == null) {
606             result = new int [2];
607             result[0] = firstValue;
608             result[1] = lastValue;
609         }
610         else {
611             int firstLength = firstExpansion==null? 1 : firstExpansion.length;
612             int lastLength = lastExpansion==null? 1 : lastExpansion.length;
613             result = new int [firstLength + lastLength];
614
615             if (firstExpansion == null) {

```

```

616         result[0] = firstValue;
617     }
618     else {
619         System.arraycopy(firstExpansion, 0, result, 0, firstLength);
620     }
621
622     if (lastExpansion == null) {
623         result[firstLength] = lastValue;
624     }
625     else {
626         System.arraycopy(lastExpansion, 0, result, firstLength, lastLength);
627     }
628 }
629
630 return result;
631 }
632
633 /**
634  * Check if a comparison order is ignorable.
635  * @return true if a character is ignorable, false otherwise.
636  */
637 final static boolean isIgnorable(int order)
638 {
639     return ((primaryOrder(order) == 0) ? true : false);
640 }
641
642 /**
643  * Get the ordering priority of the next contracting character in the
644  * string.
645  * @param ch the starting character of a contracting character token
646  * @return the next contracting character's ordering. Returns NULLORDER
647  * if the end of string is reached.
648  */
649 private int nextContractChar(int ch)
650 {
651     // First get the ordering of this single character,
652     // which is always the first element in the list
653     Vector<EntryPair> list = ordering.getContractValues(ch);
654     EntryPair pair = list.firstElement();
655     int order = pair.value;
656
657     // find out the length of the longest contracting character sequence in the list.
658     // There's logic in the builder code to make sure the longest sequence is always
659     // the last.
660     pair = list.lastElement();
661     int maxLength = pair.entryName.length();
662
663     // (the Normalizer is cloned here so that the seeking we do in the next loop
664     // won't affect our real position in the text)
665     NormalizerBase tempText = (NormalizerBase)text.clone();
666
667     // extract the next maxLength characters in the string (we have to do this using the
668     // Normalizer to ensure that our offsets correspond to those the rest of the

```

```

669     // iterator is using) and store it in "fragment".
670     tempText.previous();
671     key.setLength(0);
672     int c = tempText.next();
673     while (maxLength > 0 && c != NormalizerBase.DONE) {
674         if (Character.isSupplementaryCodePoint(c)) {
675             key.append(Character.toChars(c));
676             maxLength -= 2;
677         } else {
678             key.append((char)c);
679             --maxLength;
680         }
681         c = tempText.next();
682     }
683     String fragment = key.toString();
684     // now that we have that fragment, iterate through this list looking for the
685     // longest sequence that matches the characters in the actual text. (maxLength
686     // is used here to keep track of the length of the longest sequence)
687     // Upon exit from this loop, maxLength will contain the length of the matching
688     // sequence and order will contain the collation-element value corresponding
689     // to this sequence
690     maxLength = 1;
691     for (int i = list.size() - 1; i > 0; i--) {
692         pair = list.elementAt(i);
693         if (!pair.fwd)
694             continue;
695
696         if (fragment.startsWith(pair.entryName) && pair.entryName.length()
697             > maxLength) {
698             maxLength = pair.entryName.length();
699             order = pair.value;
700         }
701     }
702
703     // seek our current iteration position to the end of the matching sequence
704     // and return the appropriate collation-element value (if there was no matching
705     // sequence, we're already seeked to the right position and order already contains
706     // the correct collation-element value for the single character)
707     while (maxLength > 1) {
708         c = text.next();
709         maxLength -= Character.charCount(c);
710     }
711     return order;
712 }
713
714 /**
715  * Get the ordering priority of the previous contracting character in the
716  * string.
717  * @param ch the starting character of a contracting character token
718  * @return the next contracting character's ordering. Returns NULLORDER
719  * if the end of string is reached.
720  */
721 private int prevContractChar(int ch)

```

```

722     {
723         // This function is identical to nextContractChar(), except that we've
724         // switched things so that the next() and previous() calls on the Normalizer
725         // are switched and so that we skip entry pairs with the fwd flag turned on
726         // rather than off. Notice that we still use append() and startsWith() when
727         // working on the fragment. This is because the entry pairs that are used
728         // in reverse iteration have their names reversed already.
729         Vector<EntryPair> list = ordering.getContractValues(ch);
730         EntryPair pair = list.firstElement();
731         int order = pair.value;
732
733         pair = list.lastElement();
734         int maxLength = pair.entryName.length();
735
736         NormalizerBase tempText = (NormalizerBase)text.clone();
737
738         tempText.next();
739         key.setLength(0);
740         int c = tempText.previous();
741         while (maxLength > 0 && c != NormalizerBase.DONE) {
742             if (Character.isSupplementaryCodePoint(c)) {
743                 key.append(Character.toChars(c));
744                 maxLength -= 2;
745             } else {
746                 key.append((char)c);
747                 --maxLength;
748             }
749             c = tempText.previous();
750         }
751         String fragment = key.toString();
752
753         maxLength = 1;
754         for (int i = list.size() - 1; i > 0; i--) {
755             pair = list.elementAt(i);
756             if (pair.fwd)
757                 continue;
758
759             if (fragment.startsWith(pair.entryName) && pair.entryName.length()
760                 > maxLength) {
761                 maxLength = pair.entryName.length();
762                 order = pair.value;
763             }
764         }
765
766         while (maxLength > 1) {
767             c = text.previous();
768             maxLength -= Character.charCount(c);
769         }
770         return order;
771     }
772
773     final static int UNMAPPEDCHARVALUE = 0x7FFF0000;
774

```

```

775     private NormalizerBase text = null;
776     private int[] buffer = null;
777     private int expIndex = 0;
778     private StringBuffer key = new StringBuffer(5);
779     private int swapOrder = 0;
780     private RBCollationTables ordering;
781     private RuleBasedCollator owner;
782 }

```

2.11 CollationKey.java

Secuencia 20: java/text/CollationKey.java

```

1  /*
2  * Copyright (c) 1997, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright Taligent, Inc. 1996 - All Rights Reserved
28 * (C) Copyright IBM Corp. 1996 - All Rights Reserved
29 *
30 * The original version of this source code and documentation is copyrighted
31 * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32 * materials are provided under terms of a License Agreement between Taligent
33 * and Sun. This technology is protected by multiple US and International
34 * patents. This notice and attribution to Taligent may not be removed.
35 * Taligent is a registered trademark of Taligent, Inc.
36 *
37 */
38
39 package java.text;
40
41 /**

```



```
42 * A CollationKey represents a String under the
43 * rules of a specific Collator object. Comparing two
44 * CollationKeys returns the relative order of the
45 * Strings they represent. Using CollationKeys
46 * to compare Strings is generally faster than using
47 * Collator.compare. Thus, when the Strings
48 * must be compared multiple times, for example when sorting a list
49 * of Strings. It's more efficient to use CollationKeys.
50 *
51 * <p>
52 * You can not create CollationKeys directly. Rather,
53 * generate them by calling Collator.getCollationKey.
54 * You can only compare CollationKeys generated from
55 * the same Collator object.
56 *
57 * <p>
58 * Generating a CollationKey for a String
59 * involves examining the entire String
60 * and converting it to series of bits that can be compared bitwise. This
61 * allows fast comparisons once the keys are generated. The cost of generating
62 * keys is recouped in faster comparisons when Strings need
63 * to be compared many times. On the other hand, the result of a comparison
64 * is often determined by the first couple of characters of each String.
65 * Collator.compare examines only as many characters as it needs which
66 * allows it to be faster when doing single comparisons.
67 * <p>
68 * The following example shows how CollationKeys might be used
69 * to sort a list of Strings.
70 * <blockquote>
71 * <pre>{@code
72 * // Create an array of CollationKeys for the Strings to be sorted.
73 * Collator myCollator = Collator.getInstance();
74 * CollationKey[] keys = new CollationKey[3];
75 * keys[0] = myCollator.getCollationKey("Tom");
76 * keys[1] = myCollator.getCollationKey("Dick");
77 * keys[2] = myCollator.getCollationKey("Harry");
78 * sort(keys);
79 *
80 * //...
81 *
82 * // Inside body of sort routine, compare keys this way
83 * if (keys[i].compareTo(keys[j]) > 0)
84 *     // swap keys[i] and keys[j]
85 *
86 * //...
87 *
88 * // Finally, when we've returned from sort.
89 * System.out.println(keys[0].getSourceString());
90 * System.out.println(keys[1].getSourceString());
91 * System.out.println(keys[2].getSourceString());
92 * }</pre>
93 * </blockquote>
94 *
```

```

95  * @see      Collator
96  * @see      RuleBasedCollator
97  * @author    Helena Shih
98  */
99
100 public abstract class CollationKey implements Comparable<CollationKey> {
101     /**
102      * Compare this CollationKey to the target CollationKey. The collation rules of the
103      * Collator object which created these keys are applied. <strong>Note:</strong>
104      * CollationKeys created by different Collators can not be compared.
105      * @param target target CollationKey
106      * @return Returns an integer value. Value is less than zero if this is less
107      *         than target, value is zero if this and target are equal and value is greater than
108      *         zero if this is greater than target.
109      * @see java.text.Collator#compare
110      */
111     abstract public int compareTo(CollationKey target);
112
113     /**
114      * Returns the String that this CollationKey represents.
115      *
116      * @return the source string of this CollationKey
117      */
118     public String getSourceString() {
119         return source;
120     }
121
122
123     /**
124      * Converts the CollationKey to a sequence of bits. If two CollationKeys
125      * could be legitimately compared, then one could compare the byte arrays
126      * for each of those keys to obtain the same result. Byte arrays are
127      * organized most significant byte first.
128      *
129      * @return a byte array representation of the CollationKey
130      */
131     abstract public byte[] toByteArray();
132
133
134     /**
135      * CollationKey constructor.
136      *
137      * @param source the source string
138      * @exception NullPointerException if {@code source} is null
139      * @since 1.6
140      */
141     protected CollationKey(String source) {
142         if (source==null){
143             throw new NullPointerException();
144         }
145         this.source = source;
146     }
147

```

```
148     final private String source;
149 }
```

2.12 Collator.java

Secuencia 21: java/text/Collator.java

```
1  /*
2  * Copyright (c) 1997, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright Taligent, Inc. 1996-1998 - All Rights Reserved
28 * (C) Copyright IBM Corp. 1996-1998 - All Rights Reserved
29 *
30 * The original version of this source code and documentation is copyrighted
31 * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32 * materials are provided under terms of a License Agreement between Taligent
33 * and Sun. This technology is protected by multiple US and International
34 * patents. This notice and attribution to Taligent may not be removed.
35 * Taligent is a registered trademark of Taligent, Inc.
36 *
37 */
38
39 package java.text;
40
41 import java.lang.ref.SoftReference;
42 import java.text.spi.CollatorProvider;
43 import java.util.Locale;
44 import java.util.ResourceBundle;
45 import java.util.concurrent.ConcurrentHashMap;
46 import java.util.concurrent.ConcurrentMap;
47 import sun.util.locale.provider.LocaleProviderAdapter;
```

```
48 import sun.util.locale.provider.LocaleServiceProviderPool;
49
50
51 /**
52  * The Collator class performs locale-sensitive
53  * String comparison. You use this class to build
54  * searching and sorting routines for natural language text.
55  *
56  * <p>
57  * Collator is an abstract base class. Subclasses
58  * implement specific collation strategies. One subclass,
59  * RuleBasedCollator, is currently provided with
60  * the Java Platform and is applicable to a wide set of languages. Other
61  * subclasses may be created to handle more specialized needs.
62  *
63  * <p>
64  * Like other locale-sensitive classes, you can use the static
65  * factory method, getInstance, to obtain the appropriate
66  * Collator object for a given locale. You will only need
67  * to look at the subclasses of Collator if you need
68  * to understand the details of a particular collation strategy or
69  * if you need to modify that strategy.
70  *
71  * <p>
72  * The following example shows how to compare two strings using
73  * the Collator for the default locale.
74  * <blockquote>
75  * <pre>{@code
76  * // Compare two strings in the default locale
77  * Collator myCollator = Collator.getInstance();
78  * if( myCollator.compare("abc", "ABC") < 0 )
79  *     System.out.println("abc is less than ABC");
80  * else
81  *     System.out.println("abc is greater than or equal to ABC");
82  * }</pre>
83  * </blockquote>
84  *
85  * <p>
86  * You can set a Collator's strength property
87  * to determine the level of difference considered significant in
88  * comparisons. Four strengths are provided: PRIMARY,
89  * SECONDARY, TERTIARY, and IDENTICAL.
90  * The exact assignment of strengths to language features is
91  * locale dependant. For example, in Czech, "e" and "f" are considered
92  * primary differences, while "e" and "ě" are secondary differences,
93  * "e" and "E" are tertiary differences and "e" and "e" are identical.
94  * The following shows how both case and accents could be ignored for
95  * US English.
96  * <blockquote>
97  * <pre>
98  * //Get the Collator for US English and set its strength to PRIMARY
99  * Collator usCollator = Collator.getInstance(Locale.US);
100  * usCollator.setStrength(Collator.PRIMARY);
```

```

101 * if( usCollator.compare("abc", "ABC") == 0 ) {
102 *     System.out.println("Strings are equivalent");
103 * }
104 * </pre>
105 * </blockquote>
106 * <p>
107 * For comparing <code>String</code>s exactly once, the <code>compare</code>
108 * method provides the best performance. When sorting a list of
109 * <code>String</code>s however, it is generally necessary to compare each
110 * <code>String</code> multiple times. In this case, <code>CollationKey</code>s
111 * provide better performance. The <code>CollationKey</code> class converts
112 * a <code>String</code> to a series of bits that can be compared bitwise
113 * against other <code>CollationKey</code>s. A <code>CollationKey</code> is
114 * created by a <code>Collator</code> object for a given <code>String</code>.
115 * <br>
116 * <strong>Note:</strong> <code>CollationKey</code>s from different
117 * <code>Collator</code>s can not be compared. See the class description
118 * for {@link CollationKey}
119 * for an example using <code>CollationKey</code>s.
120 *
121 * @see      RuleBasedCollator
122 * @see      CollationKey
123 * @see      CollationElementIterator
124 * @see      Locale
125 * @author    Helena Shih, Laura Werner, Richard Gillam
126 */
127
128 public abstract class Collator
129     implements java.util.Comparator<Object>, Cloneable
130 {
131     /**
132     * Collator strength value. When set, only PRIMARY differences are
133     * considered significant during comparison. The assignment of strengths
134     * to language features is locale dependant. A common example is for
135     * different base letters ("a" vs "b") to be considered a PRIMARY difference.
136     * @see java.text.Collator#setStrength
137     * @see java.text.Collator#getStrength
138     */
139     public final static int PRIMARY = 0;
140     /**
141     * Collator strength value. When set, only SECONDARY and above differences are
142     * considered significant during comparison. The assignment of strengths
143     * to language features is locale dependant. A common example is for
144     * different accented forms of the same base letter ("a" vs "\u00E4") to be
145     * considered a SECONDARY difference.
146     * @see java.text.Collator#setStrength
147     * @see java.text.Collator#getStrength
148     */
149     public final static int SECONDARY = 1;
150     /**
151     * Collator strength value. When set, only TERTIARY and above differences are
152     * considered significant during comparison. The assignment of strengths
153     * to language features is locale dependant. A common example is for

```

```
154     * case differences ("a" vs "A") to be considered a TERTIARY difference.
155     * @see java.text.Collator#setStrength
156     * @see java.text.Collator#getStrength
157     */
158     public final static int TERTIARY = 2;
159
160     /**
161     * Collator strength value. When set, all differences are
162     * considered significant during comparison. The assignment of strengths
163     * to language features is locale dependant. A common example is for control
164     * characters ("&#092;u0001" vs "&#092;u0002") to be considered equal at the
165     * PRIMARY, SECONDARY, and TERTIARY levels but different at the IDENTICAL
166     * level. Additionally, differences between pre-composed accents such as
167     * "&#092;u00C0" (A-grave) and combining accents such as "A&#092;u0300"
168     * (A, combining-grave) will be considered significant at the IDENTICAL
169     * level if decomposition is set to NO_DECOMPOSITION.
170     */
171     public final static int IDENTICAL = 3;
172
173     /**
174     * Decomposition mode value. With NO_DECOMPOSITION
175     * set, accented characters will not be decomposed for collation. This
176     * is the default setting and provides the fastest collation but
177     * will only produce correct results for languages that do not use accents.
178     * @see java.text.Collator#getDecomposition
179     * @see java.text.Collator#setDecomposition
180     */
181     public final static int NO_DECOMPOSITION = 0;
182
183     /**
184     * Decomposition mode value. With CANONICAL_DECOMPOSITION
185     * set, characters that are canonical variants according to Unicode
186     * standard will be decomposed for collation. This should be used to get
187     * correct collation of accented characters.
188     * <p>
189     * CANONICAL_DECOMPOSITION corresponds to Normalization Form D as
190     * described in
191     * <a href="http://www.unicode.org/unicode/reports/tr15/tr15-23.html">Unicode
192     * Technical Report #15</a>.
193     * @see java.text.Collator#getDecomposition
194     * @see java.text.Collator#setDecomposition
195     */
196     public final static int CANONICAL_DECOMPOSITION = 1;
197
198     /**
199     * Decomposition mode value. With FULL_DECOMPOSITION
200     * set, both Unicode canonical variants and Unicode compatibility variants
201     * will be decomposed for collation. This causes not only accented
202     * characters to be collated, but also characters that have special formats
203     * to be collated with their normal form. For example, the half-width and
204     * full-width ASCII and Katakana characters are then collated together.
205     * FULL_DECOMPOSITION is the most complete and therefore the slowest
206     * decomposition mode.
```

```

207 * <p>
208 * FULL_DECOMPOSITION corresponds to Normalization Form KD as
209 * described in
210 * <a href="http://www.unicode.org/unicode/reports/tr15/tr15-23.html">Unicode
211 * Technical Report #15</a>.
212 * @see java.text.Collator#getDecomposition
213 * @see java.text.Collator#setDecomposition
214 */
215 public final static int FULL_DECOMPOSITION = 2;
216
217 /**
218 * Gets the Collator for the current default locale.
219 * The default locale is determined by java.util.Locale.getDefault.
220 * @return the Collator for the default locale.(for example, en_US)
221 * @see java.util.Locale#getDefault
222 */
223 public static synchronized Collator getInstance() {
224     return getInstance(Locale.getDefault());
225 }
226
227 /**
228 * Gets the Collator for the desired locale.
229 * @param desiredLocale the desired locale.
230 * @return the Collator for the desired locale.
231 * @see java.util.Locale
232 * @see java.util.ResourceBundle
233 */
234 public static Collator getInstance(Locale desiredLocale) {
235     SoftReference<Collator> ref = cache.get(desiredLocale);
236     Collator result = (ref != null) ? ref.get() : null;
237     if (result == null) {
238         LocaleProviderAdapter adapter;
239         adapter = LocaleProviderAdapter.getAdapter(CollatorProvider.class,
240                                                     desiredLocale);
241         CollatorProvider provider = adapter.getCollatorProvider();
242         result = provider.getInstance(desiredLocale);
243         if (result == null) {
244             result = LocaleProviderAdapter.forJRE()
245                 .getCollatorProvider().getInstance(desiredLocale);
246         }
247         while (true) {
248             if (ref != null) {
249                 // Remove the empty SoftReference if any
250                 cache.remove(desiredLocale, ref);
251             }
252             ref = cache.putIfAbsent(desiredLocale, new SoftReference<>(result));
253             if (ref == null) {
254                 break;
255             }
256             Collator cachedColl = ref.get();
257             if (cachedColl != null) {
258                 result = cachedColl;
259                 break;

```

```

260         }
261     }
262 }
263     return (Collator) result.clone(); // make the world safe
264 }
265
266 /**
267  * Compares the source string to the target string according to the
268  * collation rules for this Collator. Returns an integer less than,
269  * equal to or greater than zero depending on whether the source String is
270  * less than, equal to or greater than the target string. See the Collator
271  * class description for an example of use.
272  * <p>
273  * For a one time comparison, this method has the best performance. If a
274  * given String will be involved in multiple comparisons, CollationKey.compareTo
275  * has the best performance. See the Collator class description for an example
276  * using CollationKeys.
277  * @param source the source string.
278  * @param target the target string.
279  * @return Returns an integer value. Value is less than zero if source is less than
280  * target, value is zero if source and target are equal, value is greater than zero
281  * if source is greater than target.
282  * @see java.text.CollationKey
283  * @see java.text.Collator#getCollationKey
284  */
285 public abstract int compare(String source, String target);
286
287 /**
288  * Compares its two arguments for order. Returns a negative integer,
289  * zero, or a positive integer as the first argument is less than, equal
290  * to, or greater than the second.
291  * <p>
292  * This implementation merely returns
293  * <code> compare((String)o1, (String)o2) </code>.
294  *
295  * @return a negative integer, zero, or a positive integer as the
296  *         first argument is less than, equal to, or greater than the
297  *         second.
298  * @exception ClassCastException the arguments cannot be cast to Strings.
299  * @see java.util.Comparator
300  * @since 1.2
301  */
302 @Override
303 public int compare(Object o1, Object o2) {
304     return compare((String)o1, (String)o2);
305 }
306
307 /**
308  * Transforms the String into a series of bits that can be compared bitwise
309  * to other CollationKeys. CollationKeys provide better performance than
310  * Collator.compare when Strings are involved in multiple comparisons.
311  * See the Collator class description for an example using CollationKeys.
312  * @param source the string to be transformed into a collation key.

```



```
313     * @return the CollationKey for the given String based on this Collator's collation
314     * rules. If the source String is null, a null CollationKey is returned.
315     * @see java.text.CollationKey
316     * @see java.text.Collator#compare
317     */
318     public abstract CollationKey getCollationKey(String source);
319
320     /**
321     * Convenience method for comparing the equality of two strings based on
322     * this Collator's collation rules.
323     * @param source the source string to be compared with.
324     * @param target the target string to be compared with.
325     * @return true if the strings are equal according to the collation
326     * rules. false, otherwise.
327     * @see java.text.Collator#compare
328     */
329     public boolean equals(String source, String target)
330     {
331         return (compare(source, target) == Collator.EQUAL);
332     }
333
334     /**
335     * Returns this Collator's strength property. The strength property determines
336     * the minimum level of difference considered significant during comparison.
337     * See the Collator class description for an example of use.
338     * @return this Collator's current strength property.
339     * @see java.text.Collator#setStrength
340     * @see java.text.Collator#PRIMARY
341     * @see java.text.Collator#SECONDARY
342     * @see java.text.Collator#TERTIARY
343     * @see java.text.Collator#IDENTICAL
344     */
345     public synchronized int getStrength()
346     {
347         return strength;
348     }
349
350     /**
351     * Sets this Collator's strength property. The strength property determines
352     * the minimum level of difference considered significant during comparison.
353     * See the Collator class description for an example of use.
354     * @param newStrength the new strength value.
355     * @see java.text.Collator#getStrength
356     * @see java.text.Collator#PRIMARY
357     * @see java.text.Collator#SECONDARY
358     * @see java.text.Collator#TERTIARY
359     * @see java.text.Collator#IDENTICAL
360     * @exception IllegalArgumentException If the new strength value is not one of
361     * PRIMARY, SECONDARY, TERTIARY or IDENTICAL.
362     */
363     public synchronized void setStrength(int newStrength) {
364         if ((newStrength != PRIMARY) &&
365             (newStrength != SECONDARY) &&
```

```

366         (newStrength != TERTIARY) &&
367         (newStrength != IDENTICAL)) {
368             throw new IllegalArgumentException("Incorrect comparison level.");
369         }
370         strength = newStrength;
371     }
372
373     /**
374      * Get the decomposition mode of this Collator. Decomposition mode
375      * determines how Unicode composed characters are handled. Adjusting
376      * decomposition mode allows the user to select between faster and more
377      * complete collation behavior.
378      * <p>The three values for decomposition mode are:
379      * <UL>
380      * <LI>NO_DECOMPOSITION,
381      * <LI>CANONICAL_DECOMPOSITION
382      * <LI>FULL_DECOMPOSITION.
383      * </UL>
384      * See the documentation for these three constants for a description
385      * of their meaning.
386      * @return the decomposition mode
387      * @see java.text.Collator#setDecomposition
388      * @see java.text.Collator#NO_DECOMPOSITION
389      * @see java.text.Collator#CANONICAL_DECOMPOSITION
390      * @see java.text.Collator#FULL_DECOMPOSITION
391      */
392     public synchronized int getDecomposition()
393     {
394         return decmp;
395     }
396
397     /**
398      * Set the decomposition mode of this Collator. See getDecomposition
399      * for a description of decomposition mode.
400      * @param decompositionMode the new decomposition mode.
401      * @see java.text.Collator#getDecomposition
402      * @see java.text.Collator#NO_DECOMPOSITION
403      * @see java.text.Collator#CANONICAL_DECOMPOSITION
404      * @see java.text.Collator#FULL_DECOMPOSITION
405      * @exception IllegalArgumentException If the given value is not a valid decomposition
406      * mode.
407      */
408     public synchronized void setDecomposition(int decompositionMode) {
409         if ((decompositionMode != NO_DECOMPOSITION) &&
410             (decompositionMode != CANONICAL_DECOMPOSITION) &&
411             (decompositionMode != FULL_DECOMPOSITION)) {
412             throw new IllegalArgumentException("Wrong decomposition mode.");
413         }
414         decmp = decompositionMode;
415     }
416
417     /**
418      * Returns an array of all locales for which the
419      * <code>getInstance</code> methods of this class can return

```

```

419     * localized instances.
420     * The returned array represents the union of locales supported
421     * by the Java runtime and by installed
422     * {@link java.text.spi.CollatorProvider CollatorProvider} implementations.
423     * It must contain at least a Locale instance equal to
424     * {@link java.util.Locale#US Locale.US}.
425     *
426     * @return An array of locales for which localized
427     *         <code>Collator</code> instances are available.
428     */
429     public static synchronized Locale[] getAvailableLocales() {
430         LocaleServiceProviderPool pool =
431             LocaleServiceProviderPool.getPool(CollatorProvider.class);
432         return pool.getAvailableLocales();
433     }
434
435     /**
436     * Overrides Cloneable
437     */
438     @Override
439     public Object clone()
440     {
441         try {
442             return (Collator)super.clone();
443         } catch (CloneNotSupportedException e) {
444             throw new InternalError(e);
445         }
446     }
447
448     /**
449     * Compares the equality of two Collators.
450     * @param that the Collator to be compared with this.
451     * @return true if this Collator is the same as that Collator;
452     *         false otherwise.
453     */
454     @Override
455     public boolean equals(Object that)
456     {
457         if (this == that) {
458             return true;
459         }
460         if (that == null) {
461             return false;
462         }
463         if (getClass() != that.getClass()) {
464             return false;
465         }
466         Collator other = (Collator) that;
467         return ((strength == other.strength) &&
468             (decmp == other.decmp));
469     }
470
471     /**

```

```

472     * Generates the hash code for this Collator.
473     */
474     @Override
475     abstract public int hashCode();
476
477     /**
478     * Default constructor. This constructor is
479     * protected so subclasses can get access to it. Users typically create
480     * a Collator sub-class by calling the factory method getInstance.
481     * @see java.text.Collator#getInstance
482     */
483     protected Collator()
484     {
485         strength = TERTIARY;
486         decmp = CANONICAL_DECOMPOSITION;
487     }
488
489     private int strength = 0;
490     private int decmp = 0;
491     private static final ConcurrentMap<Locale, SoftReference<Collator>> cache
492         = new ConcurrentHashMap<>();
493
494     //
495     // FIXME: These three constants should be removed.
496     //
497     /**
498     * LESS is returned if source string is compared to be less than target
499     * string in the compare() method.
500     * @see java.text.Collator#compare
501     */
502     final static int LESS = -1;
503     /**
504     * EQUAL is returned if source string is compared to be equal to target
505     * string in the compare() method.
506     * @see java.text.Collator#compare
507     */
508     final static int EQUAL = 0;
509     /**
510     * GREATER is returned if source string is compared to be greater than
511     * target string in the compare() method.
512     * @see java.text.Collator#compare
513     */
514     final static int GREATER = 1;
515 }

```

2.13 DateFormat.java

Secuencia 22: java/text/DateFormat.java

```

1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *

```

```
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27  * (C) Copyright Taligent, Inc. 1996 - All Rights Reserved
28  * (C) Copyright IBM Corp. 1996 - All Rights Reserved
29  *
30  * The original version of this source code and documentation is copyrighted
31  * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32  * materials are provided under terms of a License Agreement between Taligent
33  * and Sun. This technology is protected by multiple US and International
34  * patents. This notice and attribution to Taligent may not be removed.
35  * Taligent is a registered trademark of Taligent, Inc.
36  *
37  */
38
39 package java.text;
40
41 import java.io.InvalidObjectException;
42 import java.text.spi.DateFormatProvider;
43 import java.util.Calendar;
44 import java.util.Date;
45 import java.util.GregorianCalendar;
46 import java.util.HashMap;
47 import java.util.Locale;
48 import java.util.Map;
49 import java.util.MissingResourceException;
50 import java.util.ResourceBundle;
51 import java.util.TimeZone;
52 import java.util.spi.LocaleServiceProvider;
53 import sun.util.locale.provider.LocaleProviderAdapter;
54 import sun.util.locale.provider.LocaleServiceProviderPool;
55
56 /**
57  * {@code DateFormat} is an abstract class for date/time formatting subclasses which
58  * formats and parses dates or time in a language-independent manner.
```

```

59 * The date/time formatting subclass, such as {@link SimpleDateFormat}, allows for
60 * formatting (i.e., date &rarr; text), parsing (text &rarr; date), and
61 * normalization. The date is represented as a <code>Date</code> object or
62 * as the milliseconds since January 1, 1970, 00:00:00 GMT.
63 *
64 * <p>{@code DateFormat} provides many class methods for obtaining default date/time
65 * formatters based on the default or a given locale and a number of formatting
66 * styles. The formatting styles include {@link #FULL}, {@link #LONG}, {@link #MEDIUM}, and {@link
    #SHORT}. More
67 * detail and examples of using these styles are provided in the method
68 * descriptions.
69 *
70 * <p>{@code DateFormat} helps you to format and parse dates for any locale.
71 * Your code can be completely independent of the locale conventions for
72 * months, days of the week, or even the calendar format: lunar vs. solar.
73 *
74 * <p>To format a date for the current Locale, use one of the
75 * static factory methods:
76 * <blockquote>
77 * <pre>{@code
78 * myString = DateFormat.getDateInstance().format(myDate);
79 * }</pre>
80 * </blockquote>
81 * <p>If you are formatting multiple dates, it is
82 * more efficient to get the format and use it multiple times so that
83 * the system doesn't have to fetch the information about the local
84 * language and country conventions multiple times.
85 * <blockquote>
86 * <pre>{@code
87 * DateFormat df = DateFormat.getDateInstance();
88 * for (int i = 0; i < myDate.length; ++i) {
89 *     output.println(df.format(myDate[i]) + " ");
90 * }
91 * }</pre>
92 * </blockquote>
93 * <p>To format a date for a different Locale, specify it in the
94 * call to {@link #getDateInstance(int, Locale) getDateInstance()}.
95 * <blockquote>
96 * <pre>{@code
97 * DateFormat df = DateFormat.getDateInstance(DateFormat.LONG, Locale.FRANCE);
98 * }</pre>
99 * </blockquote>
100 * <p>You can use a DateFormat to parse also.
101 * <blockquote>
102 * <pre>{@code
103 * myDate = df.parse(myString);
104 * }</pre>
105 * </blockquote>
106 * <p>Use {@code getDateInstance} to get the normal date format for that country.
107 * There are other static factory methods available.
108 * Use {@code getTimeInstance} to get the time format for that country.
109 * Use {@code getDateTimeInstance} to get a date and time format. You can pass in
110 * different options to these factory methods to control the length of the

```

```

111 * result; from {@link #SHORT} to {@link #MEDIUM} to {@link #LONG} to {@link #FULL}. The exact
112     result depends
113 * on the locale, but generally:
114 * <ul><li>{@link #SHORT} is completely numeric, such as {@code 12.13.52} or {@code 3:30pm}
115 * <li>{@link #MEDIUM} is longer, such as {@code Jan 12, 1952}
116 * <li>{@link #LONG} is longer, such as {@code January 12, 1952} or {@code 3:30:32pm}
117 * <li>{@link #FULL} is pretty completely specified, such as
118 *   {@code Tuesday, April 12, 1952 AD or 3:30:42pm PST}.
119 * </ul>
120 *
121 * <p>You can also set the time zone on the format if you wish.
122 * If you want even more control over the format or parsing,
123 * (or want to give your users more control),
124 * you can try casting the {@code DateFormat} you get from the factory methods
125 * to a {@link SimpleDateFormat}. This will work for the majority
126 * of countries; just remember to put it in a {@code try} block in case you
127 * encounter an unusual one.
128 *
129 * <p>You can also use forms of the parse and format methods with
130 * {@link ParsePosition} and {@link FieldPosition} to
131 * allow you to
132 * <ul><li>progressively parse through pieces of a string.
133 * <li>align any particular field, or find out where it is for selection
134 * on the screen.
135 * </ul>
136 *
137 * <h3><a name="synchronization">Synchronization</a></h3>
138 *
139 * <p>
140 * Date formats are not synchronized.
141 * It is recommended to create separate format instances for each thread.
142 * If multiple threads access a format concurrently, it must be synchronized
143 * externally.
144 *
145 * @see      Format
146 * @see      NumberFormat
147 * @see      SimpleDateFormat
148 * @see      java.util.Calendar
149 * @see      java.util.GregorianCalendar
150 * @see      java.util.TimeZone
151 * @author    Mark Davis, Chen-Lieh Huang, Alan Liu
152 */
153
154 public abstract class DateFormat extends Format {
155     /**
156      * The {@link Calendar} instance used for calculating the date-time fields
157      * and the instant of time. This field is used for both formatting and
158      * parsing.
159      *
160      * <p>Subclasses should initialize this field to a {@link Calendar}
161      * appropriate for the {@link Locale} associated with this
162      * <code>DateFormat</code>.
163      *
164      * @serial

```

```
163     */
164     protected Calendar calendar;
165
166     /**
167      * The number formatter that <code>DateFormat</code> uses to format numbers
168      * in dates and times. Subclasses should initialize this to a number format
169      * appropriate for the locale associated with this <code>DateFormat</code>.
170      * @serial
171      */
172     protected NumberFormat numberFormat;
173
174     /**
175      * Useful constant for ERA field alignment.
176      * Used in FieldPosition of date/time formatting.
177      */
178     public final static int ERA_FIELD = 0;
179
180     /**
181      * Useful constant for YEAR field alignment.
182      * Used in FieldPosition of date/time formatting.
183      */
184     public final static int YEAR_FIELD = 1;
185
186     /**
187      * Useful constant for MONTH field alignment.
188      * Used in FieldPosition of date/time formatting.
189      */
190     public final static int MONTH_FIELD = 2;
191
192     /**
193      * Useful constant for DATE field alignment.
194      * Used in FieldPosition of date/time formatting.
195      */
196     public final static int DATE_FIELD = 3;
197
198     /**
199      * Useful constant for one-based HOUR_OF_DAY field alignment.
200      * Used in FieldPosition of date/time formatting.
201      * HOUR_OF_DAY1_FIELD is used for the one-based 24-hour clock.
202      * For example, 23:59 + 01:00 results in 24:59.
203      */
204     public final static int HOUR_OF_DAY1_FIELD = 4;
205
206     /**
207      * Useful constant for zero-based HOUR_OF_DAY field alignment.
208      * Used in FieldPosition of date/time formatting.
209      * HOUR_OF_DAY0_FIELD is used for the zero-based 24-hour clock.
210      * For example, 23:59 + 01:00 results in 00:59.
211      */
212     public final static int HOUR_OF_DAY0_FIELD = 5;
213
214     /**
215      * Useful constant for MINUTE field alignment.
216      * Used in FieldPosition of date/time formatting.
217      */
218     public final static int MINUTE_FIELD = 6;
219
220     /**
221      * Useful constant for SECOND field alignment.
222      * Used in FieldPosition of date/time formatting.
223      */
224     public final static int SECOND_FIELD = 7;
```



```
216     */
217     public final static int SECOND_FIELD = 7;
218     /**
219      * Useful constant for MILLISECOND field alignment.
220      * Used in FieldPosition of date/time formatting.
221      */
222     public final static int MILLISECOND_FIELD = 8;
223     /**
224      * Useful constant for DAY_OF_WEEK field alignment.
225      * Used in FieldPosition of date/time formatting.
226      */
227     public final static int DAY_OF_WEEK_FIELD = 9;
228     /**
229      * Useful constant for DAY_OF_YEAR field alignment.
230      * Used in FieldPosition of date/time formatting.
231      */
232     public final static int DAY_OF_YEAR_FIELD = 10;
233     /**
234      * Useful constant for DAY_OF_WEEK_IN_MONTH field alignment.
235      * Used in FieldPosition of date/time formatting.
236      */
237     public final static int DAY_OF_WEEK_IN_MONTH_FIELD = 11;
238     /**
239      * Useful constant for WEEK_OF_YEAR field alignment.
240      * Used in FieldPosition of date/time formatting.
241      */
242     public final static int WEEK_OF_YEAR_FIELD = 12;
243     /**
244      * Useful constant for WEEK_OF_MONTH field alignment.
245      * Used in FieldPosition of date/time formatting.
246      */
247     public final static int WEEK_OF_MONTH_FIELD = 13;
248     /**
249      * Useful constant for AM_PM field alignment.
250      * Used in FieldPosition of date/time formatting.
251      */
252     public final static int AM_PM_FIELD = 14;
253     /**
254      * Useful constant for one-based HOUR field alignment.
255      * Used in FieldPosition of date/time formatting.
256      * HOUR1_FIELD is used for the one-based 12-hour clock.
257      * For example, 11:30 PM + 1 hour results in 12:30 AM.
258      */
259     public final static int HOUR1_FIELD = 15;
260     /**
261      * Useful constant for zero-based HOUR field alignment.
262      * Used in FieldPosition of date/time formatting.
263      * HOURO_FIELD is used for the zero-based 12-hour clock.
264      * For example, 11:30 PM + 1 hour results in 00:30 AM.
265      */
266     public final static int HOURO_FIELD = 16;
267     /**
268      * Useful constant for TIMEZONE field alignment.
```

```

269     * Used in FieldPosition of date/time formatting.
270     */
271     public final static int TIMEZONE_FIELD = 17;
272
273     // Proclaim serial compatibility with 1.1 FCS
274     private static final long serialVersionUID = 7218322306649953788L;
275
276     /**
277     * Overrides Format.
278     * Formats a time object into a time string. Examples of time objects
279     * are a time value expressed in milliseconds and a Date object.
280     * @param obj must be a Number or a Date.
281     * @param toAppendTo the string buffer for the returning time string.
282     * @return the string buffer passed in as toAppendTo, with formatted text appended.
283     * @param fieldPosition keeps track of the position of the field
284     * within the returned string.
285     * On input: an alignment field,
286     * if desired. On output: the offsets of the alignment field. For
287     * example, given a time text "1996.07.10 AD at 15:08:56 PDT",
288     * if the given fieldPosition is DateFormat.YEAR_FIELD, the
289     * begin index and end index of fieldPosition will be set to
290     * 0 and 4, respectively.
291     * Notice that if the same time field appears
292     * more than once in a pattern, the fieldPosition will be set for the first
293     * occurrence of that time field. For instance, formatting a Date to
294     * the time string "1 PM PDT (Pacific Daylight Time)" using the pattern
295     * "h a z (zzzz)" and the alignment field DateFormat.TIMEZONE_FIELD,
296     * the begin index and end index of fieldPosition will be set to
297     * 5 and 8, respectively, for the first occurrence of the timezone
298     * pattern character 'z'.
299     * @see java.text.Format
300     */
301     public final StringBuffer format(Object obj, StringBuffer toAppendTo,
302                                   FieldPosition fieldPosition)
303     {
304         if (obj instanceof Date)
305             return format((Date)obj, toAppendTo, fieldPosition);
306         else if (obj instanceof Number)
307             return format(new Date(((Number)obj).longValue()),
308                           toAppendTo, fieldPosition);
309         else
310             throw new IllegalArgumentException("Cannot format given Object as a Date");
311     }
312
313     /**
314     * Formats a Date into a date/time string.
315     * @param date a Date to be formatted into a date/time string.
316     * @param toAppendTo the string buffer for the returning date/time string.
317     * @param fieldPosition keeps track of the position of the field
318     * within the returned string.
319     * On input: an alignment field,
320     * if desired. On output: the offsets of the alignment field. For
321     * example, given a time text "1996.07.10 AD at 15:08:56 PDT",

```

```

322     * if the given fieldPosition is DateFormat.YEAR_FIELD, the
323     * begin index and end index of fieldPosition will be set to
324     * 0 and 4, respectively.
325     * Notice that if the same time field appears
326     * more than once in a pattern, the fieldPosition will be set for the first
327     * occurrence of that time field. For instance, formatting a Date to
328     * the time string "1 PM PDT (Pacific Daylight Time)" using the pattern
329     * "h a z (zzzz)" and the alignment field DateFormat.TIMEZONE_FIELD,
330     * the begin index and end index of fieldPosition will be set to
331     * 5 and 8, respectively, for the first occurrence of the timezone
332     * pattern character 'z'.
333     * @return the string buffer passed in as toAppendTo, with formatted text appended.
334     */
335     public abstract StringBuffer format(Date date, StringBuffer toAppendTo,
336                                     FieldPosition fieldPosition);
337
338     /**
339     * Formats a Date into a date/time string.
340     * @param date the time value to be formatted into a time string.
341     * @return the formatted time string.
342     */
343     public final String format(Date date)
344     {
345         return format(date, new StringBuffer(),
346                     DontCareFieldPosition.INSTANCE).toString();
347     }
348
349     /**
350     * Parses text from the beginning of the given string to produce a date.
351     * The method may not use the entire text of the given string.
352     * <p>
353     * See the {@link #parse(String, ParsePosition)} method for more information
354     * on date parsing.
355     *
356     * @param source A <code>String</code> whose beginning should be parsed.
357     * @return A <code>Date</code> parsed from the string.
358     * @exception ParseException if the beginning of the specified string
359     *         cannot be parsed.
360     */
361     public Date parse(String source) throws ParseException
362     {
363         ParsePosition pos = new ParsePosition(0);
364         Date result = parse(source, pos);
365         if (pos.index == 0)
366             throw new ParseException("Unparseable date: \"" + source + "\"",
367                                     pos.errorIndex);
368         return result;
369     }
370
371     /**
372     * Parse a date/time string according to the given parse position. For
373     * example, a time text {@code "07/10/96 4:5 PM, PDT"} will be parsed into a {@code Date}
374     * that is equivalent to {@code Date(837039900000L)}.

```

```

375 *
376 * <p> By default, parsing is lenient: If the input is not in the form used
377 * by this object's format method but can still be parsed as a date, then
378 * the parse succeeds. Clients may insist on strict adherence to the
379 * format by calling {@link #setLenient(boolean) setLenient(false)}.
380 *
381 * <p>This parsing operation uses the {@link #calendar} to produce
382 * a {@code Date}. As a result, the {@code calendar}'s date-time
383 * fields and the {@code TimeZone} value may have been
384 * overwritten, depending on subclass implementations. Any {@code
385 * TimeZone} value that has previously been set by a call to
386 * {@link #setTimeZone(java.util.TimeZone) setTimeZone} may need
387 * to be restored for further operations.
388 *
389 * @param source The date/time string to be parsed
390 *
391 * @param pos On input, the position at which to start parsing; on
392 *            output, the position at which parsing terminated, or the
393 *            start position if the parse failed.
394 *
395 * @return A {@code Date}, or {@code null} if the input could not be parsed
396 */
397 public abstract Date parse(String source, ParsePosition pos);
398
399 /**
400 * Parses text from a string to produce a <code>Date</code>.
401 * <p>
402 * The method attempts to parse text starting at the index given by
403 * <code>pos</code>.
404 * If parsing succeeds, then the index of <code>pos</code> is updated
405 * to the index after the last character used (parsing does not necessarily
406 * use all characters up to the end of the string), and the parsed
407 * date is returned. The updated <code>pos</code> can be used to
408 * indicate the starting point for the next call to this method.
409 * If an error occurs, then the index of <code>pos</code> is not
410 * changed, the error index of <code>pos</code> is set to the index of
411 * the character where the error occurred, and null is returned.
412 * <p>
413 * See the {@link #parse(String, ParsePosition)} method for more information
414 * on date parsing.
415 *
416 * @param source A <code>String</code>, part of which should be parsed.
417 * @param pos A <code>ParsePosition</code> object with index and error
418 *            index information as described above.
419 * @return A <code>Date</code> parsed from the string. In case of
420 *         error, returns null.
421 * @exception NullPointerException if <code>pos</code> is null.
422 */
423 public Object parseObject(String source, ParsePosition pos) {
424     return parse(source, pos);
425 }
426
427 /**

```

```
428     * Constant for full style pattern.
429     */
430     public static final int FULL = 0;
431     /**
432     * Constant for long style pattern.
433     */
434     public static final int LONG = 1;
435     /**
436     * Constant for medium style pattern.
437     */
438     public static final int MEDIUM = 2;
439     /**
440     * Constant for short style pattern.
441     */
442     public static final int SHORT = 3;
443     /**
444     * Constant for default style pattern. Its value is MEDIUM.
445     */
446     public static final int DEFAULT = MEDIUM;
447
448     /**
449     * Gets the time formatter with the default formatting style
450     * for the default {@link java.util.Locale.Category#FORMAT FORMAT} locale.
451     * <p>This is equivalent to calling
452     * {@link #getTimeInstance(int, Locale) getTimeInstance(DEFAULT,
453     *     Locale.getDefault(Locale.Category.FORMAT))}.
454     * @see java.util.Locale#getDefault(java.util.Locale.Category)
455     * @see java.util.Locale.Category#FORMAT
456     * @return a time formatter.
457     */
458     public final static DateFormat getTimeInstance()
459     {
460         return get(DEFAULT, 0, 1, Locale.getDefault(Locale.Category.FORMAT));
461     }
462
463     /**
464     * Gets the time formatter with the given formatting style
465     * for the default {@link java.util.Locale.Category#FORMAT FORMAT} locale.
466     * <p>This is equivalent to calling
467     * {@link #getTimeInstance(int, Locale) getTimeInstance(style,
468     *     Locale.getDefault(Locale.Category.FORMAT))}.
469     * @see java.util.Locale#getDefault(java.util.Locale.Category)
470     * @see java.util.Locale.Category#FORMAT
471     * @param style the given formatting style. For example,
472     *     SHORT for "h:mm a" in the US locale.
473     * @return a time formatter.
474     */
475     public final static DateFormat getTimeInstance(int style)
476     {
477         return get(style, 0, 1, Locale.getDefault(Locale.Category.FORMAT));
478     }
479
480     /**
```

```
481     * Gets the time formatter with the given formatting style
482     * for the given locale.
483     * @param style the given formatting style. For example,
484     * SHORT for "h:mm a" in the US locale.
485     * @param aLocale the given locale.
486     * @return a time formatter.
487     */
488     public final static DateFormat getTimeInstance(int style,
489                                                    Locale aLocale)
490     {
491         return get(style, 0, 1, aLocale);
492     }
493
494     /**
495     * Gets the date formatter with the default formatting style
496     * for the default {@link java.util.Locale.Category#FORMAT FORMAT} locale.
497     * <p>This is equivalent to calling
498     * {@link #getDateInstance(int, Locale) getDateInstance(DEFAULT,
499     *     Locale.getDefault(Locale.Category.FORMAT))}.
500     * @see java.util.Locale#getDefault(java.util.Locale.Category)
501     * @see java.util.Locale.Category#FORMAT
502     * @return a date formatter.
503     */
504     public final static DateFormat getDateInstance()
505     {
506         return get(0, DEFAULT, 2, Locale.getDefault(Locale.Category.FORMAT));
507     }
508
509     /**
510     * Gets the date formatter with the given formatting style
511     * for the default {@link java.util.Locale.Category#FORMAT FORMAT} locale.
512     * <p>This is equivalent to calling
513     * {@link #getDateInstance(int, Locale) getDateInstance(style,
514     *     Locale.getDefault(Locale.Category.FORMAT))}.
515     * @see java.util.Locale#getDefault(java.util.Locale.Category)
516     * @see java.util.Locale.Category#FORMAT
517     * @param style the given formatting style. For example,
518     * SHORT for "M/d/yy" in the US locale.
519     * @return a date formatter.
520     */
521     public final static DateFormat getDateInstance(int style)
522     {
523         return get(0, style, 2, Locale.getDefault(Locale.Category.FORMAT));
524     }
525
526     /**
527     * Gets the date formatter with the given formatting style
528     * for the given locale.
529     * @param style the given formatting style. For example,
530     * SHORT for "M/d/yy" in the US locale.
531     * @param aLocale the given locale.
532     * @return a date formatter.
533     */
```

```
534 public final static DateFormat getDateInstance(int style,
535                                                Locale aLocale)
536 {
537     return get(0, style, 2, aLocale);
538 }
539
540 /**
541  * Gets the date/time formatter with the default formatting style
542  * for the default {@link java.util.Locale.Category#FORMAT FORMAT} locale.
543  * <p>This is equivalent to calling
544  * {@link #getDateTimeInstance(int, int, Locale) getDateTimeInstance(DEFAULT,
545  *     DEFAULT, Locale.getDefault(Locale.Category.FORMAT))}.
546  * @see java.util.Locale#getDefault(java.util.Locale.Category)
547  * @see java.util.Locale.Category#FORMAT
548  * @return a date/time formatter.
549  */
550 public final static DateFormat getDateTimeInstance()
551 {
552     return get(DEFAULT, DEFAULT, 3, Locale.getDefault(Locale.Category.FORMAT));
553 }
554
555 /**
556  * Gets the date/time formatter with the given date and time
557  * formatting styles for the default {@link java.util.Locale.Category#FORMAT FORMAT} locale.
558  * <p>This is equivalent to calling
559  * {@link #getDateTimeInstance(int, int, Locale) getDateTimeInstance(dateStyle,
560  *     timeStyle, Locale.getDefault(Locale.Category.FORMAT))}.
561  * @see java.util.Locale#getDefault(java.util.Locale.Category)
562  * @see java.util.Locale.Category#FORMAT
563  * @param dateStyle the given date formatting style. For example,
564  *     SHORT for "M/d/yy" in the US locale.
565  * @param timeStyle the given time formatting style. For example,
566  *     SHORT for "h:mm a" in the US locale.
567  * @return a date/time formatter.
568  */
569 public final static DateFormat getDateTimeInstance(int dateStyle,
570                                                    int timeStyle)
571 {
572     return get(timeStyle, dateStyle, 3, Locale.getDefault(Locale.Category.FORMAT));
573 }
574
575 /**
576  * Gets the date/time formatter with the given formatting styles
577  * for the given locale.
578  * @param dateStyle the given date formatting style.
579  * @param timeStyle the given time formatting style.
580  * @param aLocale the given locale.
581  * @return a date/time formatter.
582  */
583 public final static DateFormat
584     getDateTimeInstance(int dateStyle, int timeStyle, Locale aLocale)
585 {
586     return get(timeStyle, dateStyle, 3, aLocale);
587 }
```

```
587     }
588
589     /**
590      * Get a default date/time formatter that uses the SHORT style for both the
591      * date and the time.
592      *
593      * @return a date/time formatter
594      */
595     public final static DateFormat getInstance() {
596         return getDateTimeInstance(SHORT, SHORT);
597     }
598
599     /**
600      * Returns an array of all locales for which the
601      * <code>get*Instance</code> methods of this class can return
602      * localized instances.
603      * The returned array represents the union of locales supported by the Java
604      * runtime and by installed
605      * {@link java.text.spi.DateFormatProvider DateFormatProvider} implementations.
606      * It must contain at least a <code>Locale</code> instance equal to
607      * {@link java.util.Locale#US Locale.US}.
608      *
609      * @return An array of locales for which localized
610      *         <code>DateFormat</code> instances are available.
611      */
612     public static Locale[] getAvailableLocales()
613     {
614         LocaleServiceProviderPool pool =
615             LocaleServiceProviderPool.getPool(DateFormatProvider.class);
616         return pool.getAvailableLocales();
617     }
618
619     /**
620      * Set the calendar to be used by this date format. Initially, the default
621      * calendar for the specified or default locale is used.
622      *
623      * <p>Any {@link java.util.TimeZone TimeZone} and {@linkplain
624      * #isLenient() leniency} values that have previously been set are
625      * overwritten by {@code newCalendar}'s values.
626      *
627      * @param newCalendar the new {@code Calendar} to be used by the date format
628      */
629     public void setCalendar(Calendar newCalendar)
630     {
631         this.calendar = newCalendar;
632     }
633
634     /**
635      * Gets the calendar associated with this date/time formatter.
636      *
637      * @return the calendar associated with this date/time formatter.
638      */
639     public Calendar getCalendar()
```



```
640     {
641         return calendar;
642     }
643
644     /**
645      * Allows you to set the number formatter.
646      * @param newNumberFormat the given new NumberFormat.
647      */
648     public void setNumberFormat(NumberFormat newNumberFormat)
649     {
650         this.numberFormat = newNumberFormat;
651     }
652
653     /**
654      * Gets the number formatter which this date/time formatter uses to
655      * format and parse a time.
656      * @return the number formatter which this date/time formatter uses.
657      */
658     public NumberFormat getNumberFormat()
659     {
660         return numberFormat;
661     }
662
663     /**
664      * Sets the time zone for the calendar of this {@code DateFormat} object.
665      * This method is equivalent to the following call.
666      * <blockquote><pre>{@code
667      * getCalendar().setTimeZone(zone)
668      * }</pre></blockquote>
669      *
670      * <p>The {@code TimeZone} set by this method is overwritten by a
671      * {@link #setCalendar(java.util.Calendar) setCalendar} call.
672      *
673      * <p>The {@code TimeZone} set by this method may be overwritten as
674      * a result of a call to the parse method.
675      *
676      * @param zone the given new time zone.
677      */
678     public void setTimeZone(TimeZone zone)
679     {
680         calendar.setTimeZone(zone);
681     }
682
683     /**
684      * Gets the time zone.
685      * This method is equivalent to the following call.
686      * <blockquote><pre>{@code
687      * getCalendar().getTimeZone()
688      * }</pre></blockquote>
689      *
690      * @return the time zone associated with the calendar of DateFormat.
691      */
692     public TimeZone getTimeZone()
```

```

693     {
694         return calendar.getTimeZone();
695     }
696
697     /**
698      * Specify whether or not date/time parsing is to be lenient. With
699      * lenient parsing, the parser may use heuristics to interpret inputs that
700      * do not precisely match this object's format. With strict parsing,
701      * inputs must match this object's format.
702      *
703      * <p>This method is equivalent to the following call.
704      * <blockquote><pre>{@code
705      * getCalendar().setLenient(lenient)
706      * }</pre></blockquote>
707      *
708      * <p>This leniency value is overwritten by a call to {@link
709      * #setCalendar(java.util.Calendar) setCalendar()}.
710      *
711      * @param lenient when {@code true}, parsing is lenient
712      * @see java.util.Calendar#setLenient(boolean)
713      */
714     public void setLenient(boolean lenient)
715     {
716         calendar.setLenient(lenient);
717     }
718
719     /**
720      * Tell whether date/time parsing is to be lenient.
721      * This method is equivalent to the following call.
722      * <blockquote><pre>{@code
723      * getCalendar().isLenient()
724      * }</pre></blockquote>
725      *
726      * @return {@code true} if the {@link #calendar} is lenient;
727      *         {@code false} otherwise.
728      * @see java.util.Calendar#isLenient()
729      */
730     public boolean isLenient()
731     {
732         return calendar.isLenient();
733     }
734
735     /**
736      * Overrides hashCode
737      */
738     public int hashCode() {
739         return numberFormat.hashCode();
740         // just enough fields for a reasonable distribution
741     }
742
743     /**
744      * Overrides equals
745      */

```

```

746 public boolean equals(Object obj) {
747     if (this == obj) return true;
748     if (obj == null || getClass() != obj.getClass()) return false;
749     DateFormat other = (DateFormat) obj;
750     return (// calendar.equivalentTo(other.calendar) // THIS API DOESN'T EXIST YET!
751         calendar.getFirstDayOfWeek() == other.calendar.getFirstDayOfWeek() &&
752         calendar.getMinimalDaysInFirstWeek() == other.calendar.getMinimalDaysInFirstWeek()
753             &&
754         calendar.isLenient() == other.calendar.isLenient() &&
755         calendar.getTimeZone().equals(other.calendar.getTimeZone()) &&
756         numberFormat.equals(other.numberFormat));
757 }
758 /**
759  * Overrides Cloneable
760  */
761 public Object clone()
762 {
763     DateFormat other = (DateFormat) super.clone();
764     other.calendar = (Calendar) calendar.clone();
765     other.numberFormat = (NumberFormat) numberFormat.clone();
766     return other;
767 }
768 /**
769  * Creates a DateFormat with the given time and/or date style in the given
770  * locale.
771  * @param timeStyle a value from 0 to 3 indicating the time format,
772  * ignored if flags is 2
773  * @param dateStyle a value from 0 to 3 indicating the time format,
774  * ignored if flags is 1
775  * @param flags either 1 for a time format, 2 for a date format,
776  * or 3 for a date/time format
777  * @param loc the locale for the format
778  */
779 private static DateFormat get(int timeStyle, int dateStyle,
780                               int flags, Locale loc) {
781     if ((flags & 1) != 0) {
782         if (timeStyle < 0 || timeStyle > 3) {
783             throw new IllegalArgumentException("Illegal time style " + timeStyle);
784         }
785     } else {
786         timeStyle = -1;
787     }
788     if ((flags & 2) != 0) {
789         if (dateStyle < 0 || dateStyle > 3) {
790             throw new IllegalArgumentException("Illegal date style " + dateStyle);
791         }
792     } else {
793         dateStyle = -1;
794     }
795 }
796
797 LocaleProviderAdapter adapter = LocaleProviderAdapter.getAdapter(DateFormatProvider.class,

```

```

        loc);
798     DateFormat dateFormat = get(adapter, timeStyle, dateStyle, loc);
799     if (dateFormat == null) {
800         dateFormat = get(LocaleProviderAdapter.forJRE(), timeStyle, dateStyle, loc);
801     }
802     return dateFormat;
803 }
804
805 private static DateFormat get(LocaleProviderAdapter adapter, int timeStyle, int dateStyle,
    Locale loc) {
806     DateFormatProvider provider = adapter.getDateFormatProvider();
807     DateFormat dateFormat;
808     if (timeStyle == -1) {
809         dateFormat = provider.getDateInstance(dateStyle, loc);
810     } else {
811         if (dateStyle == -1) {
812             dateFormat = provider.getTimeInstance(timeStyle, loc);
813         } else {
814             dateFormat = provider.getDateTimeInstance(dateStyle, timeStyle, loc);
815         }
816     }
817     return dateFormat;
818 }
819
820 /**
821  * Create a new date format.
822  */
823 protected DateFormat() {}
824
825 /**
826  * Defines constants that are used as attribute keys in the
827  * <code>AttributedCharacterIterator</code> returned
828  * from <code>DateFormat.formatToCharacterIterator</code> and as
829  * field identifiers in <code>FieldPosition</code>.
830  * <p>
831  * The class also provides two methods to map
832  * between its constants and the corresponding Calendar constants.
833  *
834  * @since 1.4
835  * @see java.util.Calendar
836  */
837 public static class Field extends Format.Field {
838
839     // Proclaim serial compatibility with 1.4 FCS
840     private static final long serialVersionUID = 7441350119349544720L;
841
842     // table of all instances in this class, used by readResolve
843     private static final Map<String, Field> instanceMap = new HashMap<>(18);
844     // Maps from Calendar constant (such as Calendar.ERA) to Field
845     // constant (such as Field.ERA).
846     private static final Field[] calendarToFieldMapping =
847         new Field[Calendar.FIELD_COUNT];
848

```

```

849     /** Calendar field. */
850     private int calendarField;
851
852     /**
853      * Returns the Field constant that corresponds to
854      * the Calendar constant calendarField.
855      * If there is no direct mapping between the Calendar
856      * constant and a Field, null is returned.
857      *
858      * @throws IllegalArgumentException if calendarField is
859      *         not the value of a Calendar field constant.
860      * @param calendarField Calendar field constant
861      * @return Field instance representing calendarField.
862      * @see java.util.Calendar
863      */
864     public static Field ofCalendarField(int calendarField) {
865         if (calendarField < 0 || calendarField >=
866             calendarToFieldMapping.length) {
867             throw new IllegalArgumentException("Unknown Calendar constant "
868                 + calendarField);
869         }
870         return calendarToFieldMapping[calendarField];
871     }
872
873     /**
874      * Creates a Field.
875      *
876      * @param name the name of the Field
877      * @param calendarField the Calendar constant this
878      *         Field corresponds to; any value, even one
879      *         outside the range of legal Calendar values may
880      *         be used, but -1 should be used for values
881      *         that don't correspond to legal Calendar values
882      */
883     protected Field(String name, int calendarField) {
884         super(name);
885         this.calendarField = calendarField;
886         if (this.getClass() == DateFormat.Field.class) {
887             instanceMap.put(name, this);
888             if (calendarField >= 0) {
889                 // assert(calendarField < Calendar.FIELD_COUNT);
890                 calendarToFieldMapping[calendarField] = this;
891             }
892         }
893     }
894
895     /**
896      * Returns the Calendar field associated with this
897      * attribute. For example, if this represents the hours field of
898      * a Calendar, this would return
899      * Calendar.HOUR. If there is no corresponding
900      * Calendar constant, this will return -1.
901      *

```

```
902     * @return Calendar constant for this field
903     * @see java.util.Calendar
904     */
905     public int getCalendarField() {
906         return calendarField;
907     }
908
909     /**
910     * Resolves instances being deserialized to the predefined constants.
911     *
912     * @throws InvalidObjectException if the constant could not be
913     *         resolved.
914     * @return resolved DateFormat.Field constant
915     */
916     @Override
917     protected Object readResolve() throws InvalidObjectException {
918         if (this.getClass() != DateFormat.Field.class) {
919             throw new InvalidObjectException("subclass didn't correctly implement readResolve")
920                 ;
921         }
922
923         Object instance = instanceMap.get(getName());
924         if (instance != null) {
925             return instance;
926         } else {
927             throw new InvalidObjectException("unknown attribute name");
928         }
929     }
930
931     //
932     // The constants
933     //
934     /**
935     * Constant identifying the era field.
936     */
937     public final static Field ERA = new Field("era", Calendar.ERA);
938
939     /**
940     * Constant identifying the year field.
941     */
942     public final static Field YEAR = new Field("year", Calendar.YEAR);
943
944     /**
945     * Constant identifying the month field.
946     */
947     public final static Field MONTH = new Field("month", Calendar.MONTH);
948
949     /**
950     * Constant identifying the day of month field.
951     */
952     public final static Field DAY_OF_MONTH = new
953         Field("day of month", Calendar.DAY_OF_MONTH);
```

```
954
955     /**
956      * Constant identifying the hour of day field, where the legal values
957      * are 1 to 24.
958      */
959     public final static Field HOUR_OF_DAY1 = new Field("hour of day 1",-1);
960
961     /**
962      * Constant identifying the hour of day field, where the legal values
963      * are 0 to 23.
964      */
965     public final static Field HOUR_OF_DAY0 = new
966         Field("hour of day", Calendar.HOUR_OF_DAY);
967
968     /**
969      * Constant identifying the minute field.
970      */
971     public final static Field MINUTE =new Field("minute", Calendar.MINUTE);
972
973     /**
974      * Constant identifying the second field.
975      */
976     public final static Field SECOND =new Field("second", Calendar.SECOND);
977
978     /**
979      * Constant identifying the millisecond field.
980      */
981     public final static Field MILLISECOND = new
982         Field("millisecond", Calendar.MILLISECOND);
983
984     /**
985      * Constant identifying the day of week field.
986      */
987     public final static Field DAY_OF_WEEK = new
988         Field("day of week", Calendar.DAY_OF_WEEK);
989
990     /**
991      * Constant identifying the day of year field.
992      */
993     public final static Field DAY_OF_YEAR = new
994         Field("day of year", Calendar.DAY_OF_YEAR);
995
996     /**
997      * Constant identifying the day of week field.
998      */
999     public final static Field DAY_OF_WEEK_IN_MONTH =
1000         new Field("day of week in month",
1001             Calendar.DAY_OF_WEEK_IN_MONTH);
1002
1003     /**
1004      * Constant identifying the week of year field.
1005      */
1006     public final static Field WEEK_OF_YEAR = new
```

```

1007         Field("week of year", Calendar.WEEK_OF_YEAR);
1008
1009         /**
1010          * Constant identifying the week of month field.
1011          */
1012         public final static Field WEEK_OF_MONTH = new
1013             Field("week of month", Calendar.WEEK_OF_MONTH);
1014
1015         /**
1016          * Constant identifying the time of day indicator
1017          * (e.g. "a.m." or "p.m.") field.
1018          */
1019         public final static Field AM_PM = new
1020             Field("am pm", Calendar.AM_PM);
1021
1022         /**
1023          * Constant identifying the hour field, where the legal values are
1024          * 1 to 12.
1025          */
1026         public final static Field HOUR1 = new Field("hour 1", -1);
1027
1028         /**
1029          * Constant identifying the hour field, where the legal values are
1030          * 0 to 11.
1031          */
1032         public final static Field HOURO = new
1033             Field("hour", Calendar.HOUR);
1034
1035         /**
1036          * Constant identifying the time zone field.
1037          */
1038         public final static Field TIME_ZONE = new Field("time zone", -1);
1039     }
1040 }

```

2.14 DateFormatSymbols.java

Secuencia 23: java/text/DateFormatSymbols.java

```

1  /*
2  * Copyright (c) 1996, 2016, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *

```



```
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27   * (C) Copyright Taligent, Inc. 1996 - All Rights Reserved
28   * (C) Copyright IBM Corp. 1996 - All Rights Reserved
29   *
30   * The original version of this source code and documentation is copyrighted
31   * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32   * materials are provided under terms of a License Agreement between Taligent
33   * and Sun. This technology is protected by multiple US and International
34   * patents. This notice and attribution to Taligent may not be removed.
35   * Taligent is a registered trademark of Taligent, Inc.
36   *
37   */
38
39  package java.text;
40
41  import java.io.IOException;
42  import java.io.ObjectOutputStream;
43  import java.io.Serializable;
44  import java.lang.ref.SoftReference;
45  import java.text.spi.DateFormatSymbolsProvider;
46  import java.util.Arrays;
47  import java.util.Locale;
48  import java.util.Objects;
49  import java.util.ResourceBundle;
50  import java.util.concurrent.ConcurrentHashMap;
51  import java.util.concurrent.ConcurrentMap;
52  import sun.util.locale.provider.LocaleProviderAdapter;
53  import sun.util.locale.provider.LocaleServiceProviderPool;
54  import sun.util.locale.provider.ResourceBundleBasedAdapter;
55  import sun.util.locale.provider.TimeZoneNameUtility;
56
57  /**
58   * <code>DateFormatSymbols</code> is a public class for encapsulating
59   * localizable date-time formatting data, such as the names of the
60   * months, the names of the days of the week, and the time zone data.
61   * <code>SimpleDateFormat</code> uses
62   * <code>DateFormatSymbols</code> to encapsulate this information.
63   *
64   * <p>
65   * Typically you shouldn't use <code>DateFormatSymbols</code> directly.
66   * Rather, you are encouraged to create a date-time formatter with the
67   * <code>DateFormat</code> class's factory methods: <code>getTimeInstance</code>,
68   * <code>getDateInstance</code>, or <code>getDateTimeInstance</code>.
```

```

69  * These methods automatically create a DateFormatSymbols for
70  * the formatter so that you don't have to. After the
71  * formatter is created, you may modify its format pattern using the
72  * setPattern method. For more information about
73  * creating formatters using DateFormat's factory methods,
74  * see {@link DateFormat}.
75  *
76  * <p>
77  * If you decide to create a date-time formatter with a specific
78  * format pattern for a specific locale, you can do so with:
79  * <blockquote>
80  * <pre>
81  * new SimpleDateFormat(aPattern, DateFormatSymbols.getInstance(aLocale)).
82  * </pre>
83  * </blockquote>
84  *
85  * <p>
86  * DateFormatSymbols objects are cloneable. When you obtain
87  * a DateFormatSymbols object, feel free to modify the
88  * date-time formatting data. For instance, you can replace the localized
89  * date-time format pattern characters with the ones that you feel easy
90  * to remember. Or you can change the representative cities
91  * to your favorite ones.
92  *
93  * <p>
94  * New DateFormatSymbols subclasses may be added to support
95  * SimpleDateFormat for date-time formatting for additional locales.
96
97  * @see      DateFormat
98  * @see      SimpleDateFormat
99  * @see      java.util.SimpleTimeZone
100 * @author    Chen-Lieh Huang
101 */
102 public class DateFormatSymbols implements Serializable, Cloneable {
103
104     /**
105      * Construct a DateFormatSymbols object by loading format data from
106      * resources for the default {@link java.util.Locale.Category#FORMAT FORMAT}
107      * locale. This constructor can only
108      * construct instances for the locales supported by the Java
109      * runtime environment, not for those supported by installed
110      * {@link java.text.spi.DateFormatSymbolsProvider DateFormatSymbolsProvider}
111      * implementations. For full locale coverage, use the
112      * {@link #getInstance(Locale) getInstance} method.
113      * <p>This is equivalent to calling
114      * {@link #DateFormatSymbols(Locale) DateFormatSymbols(Locale)}
115      * DateFormatSymbols(Locale.getDefault(Locale.Category.FORMAT)).
116      * @see #getInstance()
117      * @see java.util.Locale#getDefault(java.util.Locale.Category)
118      * @see java.util.Locale.Category#FORMAT
119      * @exception java.util.MissingResourceException
120      *             if the resources for the default locale cannot be
121      *             found or cannot be loaded.

```

```

122     */
123     public DateFormatSymbols()
124     {
125         initializeData(Locale.getDefault(Locale.Category.FORMAT));
126     }
127
128     /**
129      * Construct a DateFormatSymbols object by loading format data from
130      * resources for the given locale. This constructor can only
131      * construct instances for the locales supported by the Java
132      * runtime environment, not for those supported by installed
133      * {@link java.text.spi.DateFormatSymbolsProvider DateFormatSymbolsProvider}
134      * implementations. For full locale coverage, use the
135      * {@link #getInstance(Locale) getInstance} method.
136      *
137      * @param locale the desired locale
138      * @see #getInstance(Locale)
139      * @exception java.util.MissingResourceException
140      *             if the resources for the specified locale cannot be
141      *             found or cannot be loaded.
142      */
143     public DateFormatSymbols(Locale locale)
144     {
145         initializeData(locale);
146     }
147
148     /**
149      * Constructs an uninitialized DateFormatSymbols.
150      */
151     private DateFormatSymbols(boolean flag) {
152     }
153
154     /**
155      * Era strings. For example: "AD" and "BC". An array of 2 strings,
156      * indexed by Calendar.BC and Calendar.AD.
157      * @serial
158      */
159     String eras[] = null;
160
161     /**
162      * Month strings. For example: "January", "February", etc. An array
163      * of 13 strings (some calendars have 13 months), indexed by
164      * Calendar.JANUARY, Calendar.FEBRUARY, etc.
165      * @serial
166      */
167     String months[] = null;
168
169     /**
170      * Short month strings. For example: "Jan", "Feb", etc. An array of
171      * 13 strings (some calendars have 13 months), indexed by
172      * Calendar.JANUARY, Calendar.FEBRUARY, etc.
173      *
174      * @serial

```

```

175     */
176     String shortMonths[] = null;
177
178     /**
179      * Weekday strings. For example: "Sunday", "Monday", etc. An array
180      * of 8 strings, indexed by Calendar.SUNDAY,
181      * Calendar.MONDAY, etc.
182      * The element weekdays[0] is ignored.
183      * @serial
184      */
185     String weekdays[] = null;
186
187     /**
188      * Short weekday strings. For example: "Sun", "Mon", etc. An array
189      * of 8 strings, indexed by Calendar.SUNDAY,
190      * Calendar.MONDAY, etc.
191      * The element shortWeekdays[0] is ignored.
192      * @serial
193      */
194     String shortWeekdays[] = null;
195
196     /**
197      * AM and PM strings. For example: "AM" and "PM". An array of
198      * 2 strings, indexed by Calendar.AM and
199      * Calendar.PM.
200      * @serial
201      */
202     String ampm[] = null;
203
204     /**
205      * Localized names of time zones in this locale. This is a
206      * two-dimensional array of strings of size n by m,
207      * where m is at least 5. Each of the n rows is an
208      * entry containing the localized names for a single TimeZone.
209      * Each such row contains (with i ranging from
210      * 0..n-1):
211      * 


212      * - zoneStrings[i][0] - time zone ID

213      * - zoneStrings[i][1] - long name of zone in standard
214      * time

215      * - zoneStrings[i][2] - short name of zone in
216      * standard time

217      * - zoneStrings[i][3] - long name of zone in daylight
218      * saving time

219      * - zoneStrings[i][4] - short name of zone in daylight
220      * saving time

221      * 

222      * The zone ID is not localized; it's one of the valid IDs of
223      * the {@link java.util.TimeZone TimeZone} class that are not
224      * custom IDs.
225      * All other entries are localized names.
226      * @see java.util.TimeZone
227      * @serial

```

```

228     */
229     String zoneStrings[] = null;
230
231     /**
232      * Indicates that zoneStrings is set externally with setZoneStrings() method.
233      */
234     transient boolean isZoneStringsSet = false;
235
236     /**
237      * Unlocalized date-time pattern characters. For example: 'y', 'd', etc.
238      * All locales use the same these unlocalized pattern characters.
239      */
240     static final String patternChars = "GyMdkHmsSEDFwWahKzZYuXL";
241
242     static final int PATTERN_ERA = 0; // G
243     static final int PATTERN_YEAR = 1; // y
244     static final int PATTERN_MONTH = 2; // M
245     static final int PATTERN_DAY_OF_MONTH = 3; // d
246     static final int PATTERN_HOUR_OF_DAY1 = 4; // k
247     static final int PATTERN_HOUR_OF_DAY0 = 5; // H
248     static final int PATTERN_MINUTE = 6; // m
249     static final int PATTERN_SECOND = 7; // s
250     static final int PATTERN_MILLISECOND = 8; // S
251     static final int PATTERN_DAY_OF_WEEK = 9; // E
252     static final int PATTERN_DAY_OF_YEAR = 10; // D
253     static final int PATTERN_DAY_OF_WEEK_IN_MONTH = 11; // F
254     static final int PATTERN_WEEK_OF_YEAR = 12; // w
255     static final int PATTERN_WEEK_OF_MONTH = 13; // W
256     static final int PATTERN_AM_PM = 14; // a
257     static final int PATTERN_HOUR1 = 15; // h
258     static final int PATTERN_HOURO = 16; // K
259     static final int PATTERN_ZONE_NAME = 17; // z
260     static final int PATTERN_ZONE_VALUE = 18; // Z
261     static final int PATTERN_WEEK_YEAR = 19; // Y
262     static final int PATTERN_ISO_DAY_OF_WEEK = 20; // u
263     static final int PATTERN_ISO_ZONE = 21; // X
264     static final int PATTERN_MONTH_STANDALONE = 22; // L
265
266     /**
267      * Localized date-time pattern characters. For example, a locale may
268      * wish to use 'u' rather than 'y' to represent years in its date format
269      * pattern strings.
270      * This string must be exactly 18 characters long, with the index of
271      * the characters described by <code>DateFormat.ERA_FIELD</code>,
272      * <code>DateFormat.YEAR_FIELD</code>, etc. Thus, if the string were
273      * "Xz...", then localized patterns would use 'X' for era and 'z' for year.
274      * @serial
275      */
276     String localPatternChars = null;
277
278     /**
279      * The locale which is used for initializing this DateFormatSymbols object.
280      */

```

```

281     * @since 1.6
282     * @serial
283     */
284     Locale locale = null;
285
286     /* use serialVersionUID from JDK 1.1.4 for interoperability */
287     static final long serialVersionUID = -5987973545549424702L;
288
289     /**
290      * Returns an array of all locales for which the
291      * <code>getInstance</code> methods of this class can return
292      * localized instances.
293      * The returned array represents the union of locales supported by the
294      * Java runtime and by installed
295      * {@link java.text.spi.DateFormatSymbolsProvider DateFormatSymbolsProvider}
296      * implementations. It must contain at least a <code>Locale</code>
297      * instance equal to {@link java.util.Locale#US Locale.US}.
298      *
299      * @return An array of locales for which localized
300      *         <code>DateFormatSymbols</code> instances are available.
301      * @since 1.6
302      */
303     public static Locale[] getAvailableLocales() {
304         LocaleServiceProviderPool pool=
305             LocaleServiceProviderPool.getPool(DateFormatSymbolsProvider.class);
306         return pool.getAvailableLocales();
307     }
308
309     /**
310      * Gets the <code>DateFormatSymbols</code> instance for the default
311      * locale. This method provides access to <code>DateFormatSymbols</code>
312      * instances for locales supported by the Java runtime itself as well
313      * as for those supported by installed
314      * {@link java.text.spi.DateFormatSymbolsProvider DateFormatSymbolsProvider}
315      * implementations.
316      * <p>This is equivalent to calling {@link #getInstance(Locale)
317      *     getInstance(Locale.getDefault(Locale.Category.FORMAT))}.
318      * @see java.util.Locale#getDefault(java.util.Locale.Category)
319      * @see java.util.Locale.Category#FORMAT
320      * @return a <code>DateFormatSymbols</code> instance.
321      * @since 1.6
322      */
323     public static final DateFormatSymbols getInstance() {
324         return getInstance(Locale.getDefault(Locale.Category.FORMAT));
325     }
326
327     /**
328      * Gets the <code>DateFormatSymbols</code> instance for the specified
329      * locale. This method provides access to <code>DateFormatSymbols</code>
330      * instances for locales supported by the Java runtime itself as well
331      * as for those supported by installed
332      * {@link java.text.spi.DateFormatSymbolsProvider DateFormatSymbolsProvider}
333      * implementations.

```

```
334     * @param locale the given locale.
335     * @return a <code>DateFormatSymbols</code> instance.
336     * @exception NullPointerException if <code>locale</code> is null
337     * @since 1.6
338     */
339     public static final DateFormatSymbols getInstance(Locale locale) {
340         DateFormatSymbols dfs = getProviderInstance(locale);
341         if (dfs != null) {
342             return dfs;
343         }
344         throw new RuntimeException("DateFormatSymbols instance creation failed.");
345     }
346
347     /**
348     * Returns a DateFormatSymbols provided by a provider or found in
349     * the cache. Note that this method returns a cached instance,
350     * not its clone. Therefore, the instance should never be given to
351     * an application.
352     */
353     static final DateFormatSymbols getInstanceRef(Locale locale) {
354         DateFormatSymbols dfs = getProviderInstance(locale);
355         if (dfs != null) {
356             return dfs;
357         }
358         throw new RuntimeException("DateFormatSymbols instance creation failed.");
359     }
360
361     private static DateFormatSymbols getProviderInstance(Locale locale) {
362         LocaleProviderAdapter adapter = LocaleProviderAdapter.getAdapter(DateFormatSymbolsProvider.
363             class, locale);
364         DateFormatSymbolsProvider provider = adapter.getDateFormatSymbolsProvider();
365         DateFormatSymbols dfsyms = provider.getInstance(locale);
366         if (dfsyms == null) {
367             provider = LocaleProviderAdapter.forJRE().getDateFormatSymbolsProvider();
368             dfsyms = provider.getInstance(locale);
369         }
370         return dfsyms;
371     }
372
373     /**
374     * Gets era strings. For example: "AD" and "BC".
375     * @return the era strings.
376     */
377     public String[] getEras() {
378         return Arrays.copyOf(eras, eras.length);
379     }
380
381     /**
382     * Sets era strings. For example: "AD" and "BC".
383     * @param newEras the new era strings.
384     */
385     public void setEras(String[] newEras) {
386         eras = Arrays.copyOf(newEras, newEras.length);
387     }
```

```
386         cachedHashCode = 0;
387     }
388
389     /**
390     * Gets month strings. For example: "January", "February", etc.
391     *
392     * <p>If the language requires different forms for formatting and
393     * stand-alone usages, this method returns month names in the
394     * formatting form. For example, the preferred month name for
395     * January in the Czech language is <em>ledna</em> in the
396     * formatting form, while it is <em>leden</em> in the stand-alone
397     * form. This method returns {@code "ledna"} in this case. Refer
398     * to the <a href="http://unicode.org/reports/tr35/#Calendar_Elements">
399     * Calendar Elements in the Unicode Locale Data Markup Language
400     * (LDML) specification</a> for more details.
401     *
402     * @return the month strings.
403     */
404     public String[] getMonths() {
405         return Arrays.copyOf(months, months.length);
406     }
407
408     /**
409     * Sets month strings. For example: "January", "February", etc.
410     * @param newMonths the new month strings.
411     */
412     public void setMonths(String[] newMonths) {
413         months = Arrays.copyOf(newMonths, newMonths.length);
414         cachedHashCode = 0;
415     }
416
417     /**
418     * Gets short month strings. For example: "Jan", "Feb", etc.
419     *
420     * <p>If the language requires different forms for formatting and
421     * stand-alone usages, This method returns short month names in
422     * the formatting form. For example, the preferred abbreviation
423     * for January in the Catalan language is <em>de gen.</em> in the
424     * formatting form, while it is <em>gen.</em> in the stand-alone
425     * form. This method returns {@code "de gen."} in this case. Refer
426     * to the <a href="http://unicode.org/reports/tr35/#Calendar_Elements">
427     * Calendar Elements in the Unicode Locale Data Markup Language
428     * (LDML) specification</a> for more details.
429     *
430     * @return the short month strings.
431     */
432     public String[] getShortMonths() {
433         return Arrays.copyOf(shortMonths, shortMonths.length);
434     }
435
436     /**
437     * Sets short month strings. For example: "Jan", "Feb", etc.
438     * @param newShortMonths the new short month strings.
```



```
439     */
440     public void setShortMonths(String[] newShortMonths) {
441         shortMonths = Arrays.copyOf(newShortMonths, newShortMonths.length);
442         cachedHashCode = 0;
443     }
444
445     /**
446      * Gets weekday strings. For example: "Sunday", "Monday", etc.
447      * @return the weekday strings. Use Calendar.SUNDAY,
448      * Calendar.MONDAY, etc. to index the result array.
449      */
450     public String[] getWeekdays() {
451         return Arrays.copyOf(weekdays, weekdays.length);
452     }
453
454     /**
455      * Sets weekday strings. For example: "Sunday", "Monday", etc.
456      * @param newWeekdays the new weekday strings. The array should
457      * be indexed by Calendar.SUNDAY,
458      * Calendar.MONDAY, etc.
459      */
460     public void setWeekdays(String[] newWeekdays) {
461         weekdays = Arrays.copyOf(newWeekdays, newWeekdays.length);
462         cachedHashCode = 0;
463     }
464
465     /**
466      * Gets short weekday strings. For example: "Sun", "Mon", etc.
467      * @return the short weekday strings. Use Calendar.SUNDAY,
468      * Calendar.MONDAY, etc. to index the result array.
469      */
470     public String[] getShortWeekdays() {
471         return Arrays.copyOf(shortWeekdays, shortWeekdays.length);
472     }
473
474     /**
475      * Sets short weekday strings. For example: "Sun", "Mon", etc.
476      * @param newShortWeekdays the new short weekday strings. The array should
477      * be indexed by Calendar.SUNDAY,
478      * Calendar.MONDAY, etc.
479      */
480     public void setShortWeekdays(String[] newShortWeekdays) {
481         shortWeekdays = Arrays.copyOf(newShortWeekdays, newShortWeekdays.length);
482         cachedHashCode = 0;
483     }
484
485     /**
486      * Gets ampm strings. For example: "AM" and "PM".
487      * @return the ampm strings.
488      */
489     public String[] getAmPmStrings() {
490         return Arrays.copyOf(ampms, ampms.length);
491     }
```

```

492
493 /**
494  * Sets ampm strings. For example: "AM" and "PM".
495  * @param newAmps the new ampm strings.
496  */
497 public void setAmPmStrings(String[] newAmps) {
498     amps = Arrays.copyOf(newAmps, newAmps.length);
499     cachedHashCode = 0;
500 }
501
502 /**
503  * Gets time zone strings. Use of this method is discouraged; use
504  * {@link java.util.TimeZone#getName() TimeZone.getName()}
505  * instead.
506  * <p>
507  * The value returned is a
508  * two-dimensional array of strings of size <em>n</em> by <em>m</em>,
509  * where <em>m</em> is at least 5. Each of the <em>n</em> rows is an
510  * entry containing the localized names for a single <code>TimeZone</code>.
511  * Each such row contains (with <code>i</code> ranging from
512  * 0..<em>n</em>-1):
513  * <ul>
514  * <li><code>zoneStrings[i][0]</code> - time zone ID</li>
515  * <li><code>zoneStrings[i][1]</code> - long name of zone in standard
516  * time</li>
517  * <li><code>zoneStrings[i][2]</code> - short name of zone in
518  * standard time</li>
519  * <li><code>zoneStrings[i][3]</code> - long name of zone in daylight
520  * saving time</li>
521  * <li><code>zoneStrings[i][4]</code> - short name of zone in daylight
522  * saving time</li>
523  * </ul>
524  * The zone ID is <em>not</em> localized; it's one of the valid IDs of
525  * the {@link java.util.TimeZone TimeZone} class that are not
526  * <a href="..util/TimeZone.html#CustomID">custom IDs</a>.
527  * All other entries are localized names. If a zone does not implement
528  * daylight saving time, the daylight saving time names should not be used.
529  * <p>
530  * If {@link #setZoneStrings(String[][]) setZoneStrings} has been called
531  * on this <code>DateFormatSymbols</code> instance, then the strings
532  * provided by that call are returned. Otherwise, the returned array
533  * contains names provided by the Java runtime and by installed
534  * {@link java.util.spi.TimeZoneNameProvider TimeZoneNameProvider}
535  * implementations.
536  *
537  * @return the time zone strings.
538  * @see #setZoneStrings(String[][])
539  */
540 public String[][] getZoneStrings() {
541     return getZoneStringsImpl(true);
542 }
543
544 /**

```

```

545 * Sets time zone strings. The argument must be a
546 * two-dimensional array of strings of size <em>n</em> by <em>m</em>,
547 * where <em>m</em> is at least 5. Each of the <em>n</em> rows is an
548 * entry containing the localized names for a single <code>TimeZone</code>.
549 * Each such row contains (with <code>i</code> ranging from
550 * 0..<em>n</em>-1):
551 * <ul>
552 * <li><code>zoneStrings[i][0]</code> - time zone ID</li>
553 * <li><code>zoneStrings[i][1]</code> - long name of zone in standard
554 * time</li>
555 * <li><code>zoneStrings[i][2]</code> - short name of zone in
556 * standard time</li>
557 * <li><code>zoneStrings[i][3]</code> - long name of zone in daylight
558 * saving time</li>
559 * <li><code>zoneStrings[i][4]</code> - short name of zone in daylight
560 * saving time</li>
561 * </ul>
562 * The zone ID is <em>not</em> localized; it's one of the valid IDs of
563 * the {<a href="http://java.util.TimeZone.html#CustomID">link java.util.TimeZone</a> TimeZone} class that are not
564 * <a href="http://java.util.TimeZone.html#CustomID">custom IDs</a>.
565 * All other entries are localized names.
566 *
567 * @param newZoneStrings the new time zone strings.
568 * @exception IllegalArgumentException if the length of any row in
569 * <code>newZoneStrings</code> is less than 5
570 * @exception NullPointerException if <code>newZoneStrings</code> is null
571 * @see #getZoneStrings()
572 */
573 public void setZoneStrings(String[] [] newZoneStrings) {
574     String[] [] aCopy = new String[newZoneStrings.length] [];
575     for (int i = 0; i < newZoneStrings.length; ++i) {
576         int len = newZoneStrings[i].length;
577         if (len < 5) {
578             throw new IllegalArgumentException();
579         }
580         aCopy[i] = Arrays.copyOf(newZoneStrings[i], len);
581     }
582     zoneStrings = aCopy;
583     isZoneStringsSet = true;
584     cachedHashCode = 0;
585 }
586
587 /**
588 * Gets localized date-time pattern characters. For example: 'u', 't', etc.
589 * @return the localized date-time pattern characters.
590 */
591 public String getLocalPatternChars() {
592     return localPatternChars;
593 }
594
595 /**
596 * Sets localized date-time pattern characters. For example: 'u', 't', etc.
597 * @param newLocalPatternChars the new localized date-time

```

```

598     * pattern characters.
599     */
600     public void setLocalPatternChars(String newLocalPatternChars) {
601         // Call toString() to throw an NPE in case the argument is null
602         localPatternChars = newLocalPatternChars.toString();
603         cachedHashCode = 0;
604     }
605
606     /**
607     * Overrides Cloneable
608     */
609     public Object clone()
610     {
611         try
612         {
613             DateFormatSymbols other = (DateFormatSymbols)super.clone();
614             copyMembers(this, other);
615             return other;
616         } catch (CloneNotSupportedException e) {
617             throw new InternalError(e);
618         }
619     }
620
621     /**
622     * Override hashCode.
623     * Generates a hash code for the DateFormatSymbols object.
624     */
625     @Override
626     public int hashCode() {
627         int hashCode = cachedHashCode;
628         if (hashCode == 0) {
629             hashCode = 5;
630             hashCode = 11 * hashCode + Arrays.hashCode(eras);
631             hashCode = 11 * hashCode + Arrays.hashCode(months);
632             hashCode = 11 * hashCode + Arrays.hashCode(shortMonths);
633             hashCode = 11 * hashCode + Arrays.hashCode(weekdays);
634             hashCode = 11 * hashCode + Arrays.hashCode(shortWeekdays);
635             hashCode = 11 * hashCode + Arrays.hashCode(ampms);
636             hashCode = 11 * hashCode + Arrays.deepHashCode(getZoneStringsWrapper());
637             hashCode = 11 * hashCode + Objects.hashCode(localPatternChars);
638             cachedHashCode = hashCode;
639         }
640
641         return hashCode;
642     }
643
644     /**
645     * Override equals
646     */
647     public boolean equals(Object obj)
648     {
649         if (this == obj) return true;
650         if (obj == null || getClass() != obj.getClass()) return false;

```

```

651     DateFormatSymbols that = (DateFormatSymbols) obj;
652     return (Arrays.equals(eras, that.eras)
653         && Arrays.equals(months, that.months)
654         && Arrays.equals(shortMonths, that.shortMonths)
655         && Arrays.equals(weekdays, that.weekdays)
656         && Arrays.equals(shortWeekdays, that.shortWeekdays)
657         && Arrays.equals(ampms, that.ampms)
658         && Arrays.deepEquals(getZoneStringsWrapper(), that.getZoneStringsWrapper())
659         && ((localPatternChars != null
660             && localPatternChars.equals(that.localPatternChars))
661             || (localPatternChars == null
662                 && that.localPatternChars == null)));
663 }
664
665 // =====privates=====
666
667 /**
668  * Useful constant for defining time zone offsets.
669  */
670 static final int millisPerHour = 60*60*1000;
671
672 /**
673  * Cache to hold DateFormatSymbols instances per Locale.
674  */
675 private static final ConcurrentMap<Locale, SoftReference<DateFormatSymbols>> cachedInstances
676     = new ConcurrentHashMap<>(3);
677
678 private transient int lastZoneIndex = 0;
679
680 /**
681  * Cached hash code
682  */
683 transient volatile int cachedHashCode = 0;
684
685 /**
686  * Initializes this DateFormatSymbols with the locale data. This method uses
687  * a cached DateFormatSymbols instance for the given locale if available. If
688  * there's no cached one, this method creates an uninitialized instance and
689  * populates its fields from the resource bundle for the locale, and caches
690  * the instance. Note: zoneStrings isn't initialized in this method.
691  */
692 private void initializeData(Locale locale) {
693     SoftReference<DateFormatSymbols> ref = cachedInstances.get(locale);
694     DateFormatSymbols dfs;
695     if (ref == null || (dfs = ref.get()) == null) {
696         if (ref != null) {
697             // Remove the empty SoftReference
698             cachedInstances.remove(locale, ref);
699         }
700         dfs = new DateFormatSymbols(false);
701
702         // Initialize the fields from the ResourceBundle for locale.
703         LocaleProviderAdapter adapter

```

```

704         = LocaleProviderAdapter.getAdapter(DateFormatSymbolsProvider.class, locale);
705         // Avoid any potential recursions
706         if (!(adapter instanceof ResourceBundleBasedAdapter)) {
707             adapter = LocaleProviderAdapter.getResourceBundleBased();
708         }
709         ResourceBundle resource
710             = ((ResourceBundleBasedAdapter)adapter).getLocaleData().getDateFormatData(locale);
711
712         dfs.locale = locale;
713         // JRE and CLDR use different keys
714         // JRE: Eras, short.Eras and narrow.Eras
715         // CLDR: long.Eras, Eras and narrow.Eras
716         if (resource.containsKey("Eras")) {
717             dfs.eras = resource.getStringArray("Eras");
718         } else if (resource.containsKey("long.Eras")) {
719             dfs.eras = resource.getStringArray("long.Eras");
720         } else if (resource.containsKey("short.Eras")) {
721             dfs.eras = resource.getStringArray("short.Eras");
722         }
723         dfs.months = resource.getStringArray("MonthNames");
724         dfs.shortMonths = resource.getStringArray("MonthAbbreviations");
725         dfs.amPms = resource.getStringArray("AmPmMarkers");
726         dfs.localPatternChars = resource.getString("DateTimePatternChars");
727
728         // Day of week names are stored in a 1-based array.
729         dfs.weekdays = toOneBasedArray(resource.getStringArray("DayNames"));
730         dfs.shortWeekdays = toOneBasedArray(resource.getStringArray("DayAbbreviations"));
731
732         // Put dfs in the cache
733         ref = new SoftReference<>(dfs);
734         SoftReference<DateFormatSymbols> x = cachedInstances.putIfAbsent(locale, ref);
735         if (x != null) {
736             DateFormatSymbols y = x.get();
737             if (y == null) {
738                 // Replace the empty SoftReference with ref.
739                 cachedInstances.replace(locale, x, ref);
740             } else {
741                 ref = x;
742                 dfs = y;
743             }
744         }
745         // If the bundle's locale isn't the target locale, put another cache
746         // entry for the bundle's locale.
747         Locale bundleLocale = resource.getLocale();
748         if (!bundleLocale.equals(locale)) {
749             SoftReference<DateFormatSymbols> z
750                 = cachedInstances.putIfAbsent(bundleLocale, ref);
751             if (z != null && z.get() == null) {
752                 cachedInstances.replace(bundleLocale, z, ref);
753             }
754         }
755     }
756

```

```

757     // Copy the field values from dfs to this instance.
758     copyMembers(dfs, this);
759 }
760
761 private static String[] toOneBasedArray(String[] src) {
762     int len = src.length;
763     String[] dst = new String[len + 1];
764     dst[0] = "";
765     for (int i = 0; i < len; i++) {
766         dst[i + 1] = src[i];
767     }
768     return dst;
769 }
770
771 /**
772  * Package private: used by SimpleDateFormat
773  * Gets the index for the given time zone ID to obtain the time zone
774  * strings for formatting. The time zone ID is just for programmatic
775  * lookup. NOT LOCALIZED!!!
776  * @param ID the given time zone ID.
777  * @return the index of the given time zone ID. Returns -1 if
778  * the given time zone ID can't be located in the DateFormatSymbols object.
779  * @see java.util.SimpleTimeZone
780  */
781 final int getZoneIndex(String ID) {
782     String[] zoneStrings = getZoneStringsWrapper();
783
784     /*
785      * getZoneIndex has been re-written for performance reasons. instead of
786      * traversing the zoneStrings array every time, we cache the last used zone
787      * index
788      */
789     if (lastZoneIndex < zoneStrings.length && ID.equals(zoneStrings[lastZoneIndex][0])) {
790         return lastZoneIndex;
791     }
792
793     /* slow path, search entire list */
794     for (int index = 0; index < zoneStrings.length; index++) {
795         if (ID.equals(zoneStrings[index][0])) {
796             lastZoneIndex = index;
797             return index;
798         }
799     }
800
801     return -1;
802 }
803
804 /**
805  * Wrapper method to the getZoneStrings(), which is called from inside
806  * the java.text package and not to mutate the returned arrays, so that
807  * it does not need to create a defensive copy.
808  */
809 final String[] getZoneStringsWrapper() {

```

```

810     if (isSubclassObject()) {
811         return getZoneStrings();
812     } else {
813         return getZoneStringsImpl(false);
814     }
815 }
816
817 private String[] [] getZoneStringsImpl(boolean needsCopy) {
818     if (zoneStrings == null) {
819         zoneStrings = TimeZoneNameUtility.getZoneStrings(locale);
820     }
821
822     if (!needsCopy) {
823         return zoneStrings;
824     }
825
826     int len = zoneStrings.length;
827     String[] [] aCopy = new String[len] [];
828     for (int i = 0; i < len; i++) {
829         aCopy[i] = Arrays.copyOf(zoneStrings[i], zoneStrings[i].length);
830     }
831     return aCopy;
832 }
833
834 private boolean isSubclassObject() {
835     return !getClass().getName().equals("java.text.DateFormatSymbols");
836 }
837
838 /**
839  * Clones all the data members from the source DateFormatSymbols to
840  * the target DateFormatSymbols.
841  *
842  * @param src the source DateFormatSymbols.
843  * @param dst the target DateFormatSymbols.
844  */
845 private void copyMembers(DateFormatSymbols src, DateFormatSymbols dst)
846 {
847     dst.locale = src.locale;
848     dst.eras = Arrays.copyOf(src.eras, src.eras.length);
849     dst.months = Arrays.copyOf(src.months, src.months.length);
850     dst.shortMonths = Arrays.copyOf(src.shortMonths, src.shortMonths.length);
851     dst.weekdays = Arrays.copyOf(src.weekdays, src.weekdays.length);
852     dst.shortWeekdays = Arrays.copyOf(src.shortWeekdays, src.shortWeekdays.length);
853     dst.ampms = Arrays.copyOf(src.ampms, src.ampms.length);
854     if (src.zoneStrings != null) {
855         dst.zoneStrings = src.getZoneStringsImpl(true);
856     } else {
857         dst.zoneStrings = null;
858     }
859     dst.localPatternChars = src.localPatternChars;
860     dst.cachedHashCode = 0;
861 }
862

```



```

863  /**
864   * Write out the default serializable data, after ensuring the
865   * <code>zoneStrings</code> field is initialized in order to make
866   * sure the backward compatibility.
867   *
868   * @since 1.6
869   */
870  private void writeObject(ObjectOutputStream stream) throws IOException {
871      if (zoneStrings == null) {
872          zoneStrings = TimeZoneNameUtility.getZoneStrings(locale);
873      }
874      stream.defaultWriteObject();
875  }
876  }

```

2.15 DecimalFormat.java

Secuencia 24: java/text/DecimalFormat.java

```

1  /*
2   * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3   * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4   *
5   *
6   *
7   *
8   *
9   *
10  *
11  *
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27   * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28   * (C) Copyright IBM Corp. 1996 - 1998 - All Rights Reserved
29   *
30   * The original version of this source code and documentation is copyrighted
31   * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32   * materials are provided under terms of a License Agreement between Taligent
33   * and Sun. This technology is protected by multiple US and International
34   * patents. This notice and attribution to Taligent may not be removed.
35   * Taligent is a registered trademark of Taligent, Inc.

```

```

36  *
37  */
38
39  package java.text;
40
41  import java.io.IOException;
42  import java.io.InvalidObjectException;
43  import java.io.ObjectInputStream;
44  import java.math.BigDecimal;
45  import java.math.BigInteger;
46  import java.math.RoundingMode;
47  import java.text.spi.NumberFormatProvider;
48  import java.util.ArrayList;
49  import java.util.Currency;
50  import java.util.Locale;
51  import java.util.ResourceBundle;
52  import java.util.concurrent.ConcurrentHashMap;
53  import java.util.concurrent.ConcurrentMap;
54  import java.util.concurrent.atomic.AtomicInteger;
55  import java.util.concurrent.atomic.AtomicLong;
56  import sun.util.locale.provider.LocaleProviderAdapter;
57  import sun.util.locale.provider.ResourceBundleBasedAdapter;
58
59  /**
60   * <code>DecimalFormat</code> is a concrete subclass of
61   * <code>NumberFormat</code> that formats decimal numbers. It has a variety of
62   * features designed to make it possible to parse and format numbers in any
63   * locale, including support for Western, Arabic, and Indic digits. It also
64   * supports different kinds of numbers, including integers (123), fixed-point
65   * numbers (123.4), scientific notation (1.23E4), percentages (12%), and
66   * currency amounts ($123). All of these can be localized.
67   *
68   * <p>To obtain a <code>NumberFormat</code> for a specific locale, including the
69   * default locale, call one of <code>NumberFormat</code>'s factory methods, such
70   * as <code>getInstance()</code>. In general, do not call the
71   * <code>DecimalFormat</code> constructors directly, since the
72   * <code>NumberFormat</code> factory methods may return subclasses other than
73   * <code>DecimalFormat</code>. If you need to customize the format object, do
74   * something like this:
75   *
76   * <blockquote><pre>
77   * NumberFormat f = NumberFormat.getInstance(loc);
78   * if (f instanceof DecimalFormat) {
79   *     ((DecimalFormat) f).setDecimalSeparatorAlwaysShown(true);
80   * }
81   * </pre></blockquote>
82   *
83   * <p>A <code>DecimalFormat</code> comprises a <em>pattern</em> and a set of
84   * <em>symbols</em>. The pattern may be set directly using
85   * <code>applyPattern()</code>, or indirectly using the API methods. The
86   * symbols are stored in a <code>DecimalFormatSymbols</code> object. When using
87   * the <code>NumberFormat</code> factory methods, the pattern and symbols are
88   * read from localized <code>ResourceBundle</code>s.

```

```

89 *
90 * <h3>Patterns</h3>
91 *
92 * <code>DecimalFormat</code> patterns have the following syntax:
93 * <blockquote><pre>
94 * <i>Pattern:</i>
95 *     <i>PositivePattern</i>
96 *     <i>PositivePattern</i> ; <i>NegativePattern</i>
97 * <i>PositivePattern:</i>
98 *     <i>Prefix<sub>opt</sub></i> <i>Number</i> <i>Suffix<sub>opt</sub></i>
99 * <i>NegativePattern:</i>
100 *     <i>Prefix<sub>opt</sub></i> <i>Number</i> <i>Suffix<sub>opt</sub></i>
101 * <i>Prefix:</i>
102 *     any Unicode characters except &#92;uFFFE, &#92;uFFFF, and special characters
103 * <i>Suffix:</i>
104 *     any Unicode characters except &#92;uFFFE, &#92;uFFFF, and special characters
105 * <i>Number:</i>
106 *     <i>Integer</i> <i>Exponent<sub>opt</sub></i>
107 *     <i>Integer</i> . <i>Fraction</i> <i>Exponent<sub>opt</sub></i>
108 * <i>Integer:</i>
109 *     <i>MinimumInteger</i>
110 *     #
111 *     # <i>Integer</i>
112 *     # , <i>Integer</i>
113 * <i>MinimumInteger:</i>
114 *     0
115 *     0 <i>MinimumInteger</i>
116 *     0 , <i>MinimumInteger</i>
117 * <i>Fraction:</i>
118 *     <i>MinimumFraction<sub>opt</sub></i> <i>OptionalFraction<sub>opt</sub></i>
119 * <i>MinimumFraction:</i>
120 *     0 <i>MinimumFraction<sub>opt</sub></i>
121 * <i>OptionalFraction:</i>
122 *     # <i>OptionalFraction<sub>opt</sub></i>
123 * <i>Exponent:</i>
124 *     E <i>MinimumExponent</i>
125 * <i>MinimumExponent:</i>
126 *     0 <i>MinimumExponent<sub>opt</sub></i>
127 * </pre></blockquote>
128 *
129 * <p>A <code>DecimalFormat</code> pattern contains a positive and negative
130 * subpattern, for example, <code>"#,##0.00;(#,##0.00)"</code>. Each
131 * subpattern has a prefix, numeric part, and suffix. The negative subpattern
132 * is optional; if absent, then the positive subpattern prefixed with the
133 * localized minus sign (<code>'-'</code> in most locales) is used as the
134 * negative subpattern. That is, <code>"0.00"</code> alone is equivalent to
135 * <code>"0.00;-0.00"</code>. If there is an explicit negative subpattern, it
136 * serves only to specify the negative prefix and suffix; the number of digits,
137 * minimal digits, and other characteristics are all the same as the positive
138 * pattern. That means that <code>"#,##0.0#;(#)"</code> produces precisely
139 * the same behavior as <code>"#,##0.0#;(#,##0.0#)"</code>.
140 *
141 * <p>The prefixes, suffixes, and various symbols used for infinity, digits,

```

```

142 * thousands separators, decimal separators, etc. may be set to arbitrary
143 * values, and they will appear properly during formatting. However, care must
144 * be taken that the symbols and strings do not conflict, or parsing will be
145 * unreliable. For example, either the positive and negative prefixes or the
146 * suffixes must be distinct for DecimalFormat.parse() to be able
147 * to distinguish positive from negative values. (If they are identical, then
148 * DecimalFormat will behave as if no negative subpattern was
149 * specified.) Another example is that the decimal separator and thousands
150 * separator should be distinct characters, or parsing will be impossible.
151 *
152 * <p>The grouping separator is commonly used for thousands, but in some
153 * countries it separates ten-thousands. The grouping size is a constant number
154 * of digits between the grouping characters, such as 3 for 100,000,000 or 4 for
155 * 1,0000,0000. If you supply a pattern with multiple grouping characters, the
156 * interval between the last one and the end of the integer is the one that is
157 * used. So "#,###,####" == "#####,####" ==
158 * "##,###,####".
159 *
160 * <h4>Special Pattern Characters</h4>
161 *
162 * <p>Many characters in a pattern are taken literally; they are matched during
163 * parsing and output unchanged during formatting. Special characters, on the
164 * other hand, stand for other characters, strings, or classes of characters.
165 * They must be quoted, unless noted otherwise, if they are to appear in the
166 * prefix or suffix as literals.
167 *
168 * <p>The characters listed here are used in non-localized patterns. Localized
169 * patterns use the corresponding characters taken from this formatter's
170 * DecimalFormatSymbols object instead, and these characters lose
171 * their special status. Two exceptions are the currency sign and quote, which
172 * are not localized.
173 *
174 * <blockquote>
175 * <table border=0 cellpadding=3 cellspacing=0 summary="Chart showing symbol,
176 * location, localized, and meaning.">
177 *   <tr style="background-color: rgb(204, 204, 255);">
178 *     <th align=left>Symbol
179 *     <th align=left>Location
180 *     <th align=left>Localized?
181 *     <th align=left>Meaning
182 *   <tr valign=top>
183 *     <td><code>0</code>
184 *     <td>Number
185 *     <td>Yes
186 *     <td>Digit
187 *   <tr style="vertical-align: top; background-color: rgb(238, 238, 255);">
188 *     <td><code>#</code>
189 *     <td>Number
190 *     <td>Yes
191 *     <td>Digit, zero shows as absent
192 *   <tr valign=top>
193 *     <td><code>.</code>
194 *     <td>Number

```

```

195 *      <td>Yes
196 *      <td>Decimal separator or monetary decimal separator
197 *      <tr style="vertical-align: top; background-color: rgb(238, 238, 255);">
198 *          <td><code>-</code>
199 *          <td>Number
200 *          <td>Yes
201 *          <td>Minus sign
202 *      <tr valign=top>
203 *          <td><code>,</code>
204 *          <td>Number
205 *          <td>Yes
206 *          <td>Grouping separator
207 *      <tr style="vertical-align: top; background-color: rgb(238, 238, 255);">
208 *          <td><code>E</code>
209 *          <td>Number
210 *          <td>Yes
211 *          <td>Separates mantissa and exponent in scientific notation.
212 *              <em>Need not be quoted in prefix or suffix.</em>
213 *      <tr valign=top>
214 *          <td><code>;</code>
215 *          <td>Subpattern boundary
216 *          <td>Yes
217 *          <td>Separates positive and negative subpatterns
218 *      <tr style="vertical-align: top; background-color: rgb(238, 238, 255);">
219 *          <td><code>%</code>
220 *          <td>Prefix or suffix
221 *          <td>Yes
222 *          <td>Multiply by 100 and show as percentage
223 *      <tr valign=top>
224 *          <td><code>&#92;u2030</code>
225 *          <td>Prefix or suffix
226 *          <td>Yes
227 *          <td>Multiply by 1000 and show as per mille value
228 *      <tr style="vertical-align: top; background-color: rgb(238, 238, 255);">
229 *          <td><code>&#164;</code> (<code>&#92;u00A4</code>)
230 *          <td>Prefix or suffix
231 *          <td>No
232 *          <td>Currency sign, replaced by currency symbol. If
233 *              doubled, replaced by international currency symbol.
234 *              If present in a pattern, the monetary decimal separator
235 *              is used instead of the decimal separator.
236 *      <tr valign=top>
237 *          <td><code>'</code>
238 *          <td>Prefix or suffix
239 *          <td>No
240 *          <td>Used to quote special characters in a prefix or suffix,
241 *              for example, <code>'#</code> formats 123 to
242 *              <code>"#123"</code>. To create a single quote
243 *              itself, use two in a row: <code>"# o'clock"</code>.
244 * </table>
245 * </blockquote>
246 *
247 * <h4>Scientific Notation</h4>

```

```

248 *
249 * <p>Numbers in scientific notation are expressed as the product of a mantissa
250 * and a power of ten, for example, 1234 can be expressed as 1.234 x 103. The
251 * mantissa is often in the range 1.0 &le; x {@literal <} 10.0, but it need not
252 * be.
253 * <code>DecimalFormat</code> can be instructed to format and parse scientific
254 * notation <em>only via a pattern</em>; there is currently no factory method
255 * that creates a scientific notation format. In a pattern, the exponent
256 * character immediately followed by one or more digit characters indicates
257 * scientific notation. Example: <code>"0.###E0"</code> formats the number
258 * 1234 as <code>"1.234E3"</code>.
259 *
260 * <ul>
261 * <li>The number of digit characters after the exponent character gives the
262 * minimum exponent digit count. There is no maximum. Negative exponents are
263 * formatted using the localized minus sign, <em>not</em> the prefix and suffix
264 * from the pattern. This allows patterns such as <code>"0.###E0 m/s"</code>.
265 *
266 * <li>The minimum and maximum number of integer digits are interpreted
267 * together:
268 *
269 * <ul>
270 * <li>If the maximum number of integer digits is greater than their minimum number
271 * and greater than 1, it forces the exponent to be a multiple of the maximum
272 * number of integer digits, and the minimum number of integer digits to be
273 * interpreted as 1. The most common use of this is to generate
274 * <em>engineering notation</em>, in which the exponent is a multiple of three,
275 * e.g., <code>"##0.#####E0"</code>. Using this pattern, the number 12345
276 * formats to <code>"12.345E3"</code>, and 123456 formats to
277 * <code>"123.456E3"</code>.
278 *
279 * <li>Otherwise, the minimum number of integer digits is achieved by adjusting the
280 * exponent. Example: 0.00123 formatted with <code>"00.###E0"</code> yields
281 * <code>"12.3E-4"</code>.
282 * </ul>
283 *
284 * <li>The number of significant digits in the mantissa is the sum of the
285 * <em>minimum integer</em> and <em>maximum fraction</em> digits, and is
286 * unaffected by the maximum integer digits. For example, 12345 formatted with
287 * <code>"##0.###E0"</code> is <code>"12.3E3"</code>. To show all digits, set
288 * the significant digits count to zero. The number of significant digits
289 * does not affect parsing.
290 *
291 * <li>Exponential patterns may not contain grouping separators.
292 * </ul>
293 *
294 * <h4>Rounding</h4>
295 *
296 * <code>DecimalFormat</code> provides rounding modes defined in
297 * {@link java.math.RoundingMode} for formatting. By default, it uses
298 * {@link java.math.RoundingMode#HALF_EVEN RoundingMode.HALF_EVEN}.
299 *
300 * <h4>Digits</h4>

```

```

301 *
302 * For formatting, DecimalFormat uses the ten consecutive
303 * characters starting with the localized zero digit defined in the
304 * DecimalFormatSymbols object as digits. For parsing, these
305 * digits as well as all Unicode decimal digits, as defined by
306 * {@link Character#digit Character.digit}, are recognized.
307 *
308 * 

#### Special Values


309 *
310 * 

NaN is formatted as a string, which typically has a single character
311 * &#92;uFFFD. This string is determined by the
312 * DecimalFormatSymbols object. This is the only value for which
313 * the prefixes and suffixes are not used.
314 *
315 * 

Infinity is formatted as a string, which typically has a single character
316 * &#92;u221E, with the positive or negative prefixes and suffixes
317 * applied. The infinity string is determined by the
318 * DecimalFormatSymbols object.
319 *
320 * 

Negative zero ("-0") parses to
321 * 


322 * - BigDecimal(0) if isParseBigDecimal() is
323 * true,
324 * - Long(0) if isParseBigDecimal() is false
325 * and isParseIntegerOnly() is true,
326 * - Double(-0.0) if both isParseBigDecimal()
327 * and isParseIntegerOnly() are false.
328 *

329 *


330 * 

#### Synchronization


331 *
332 * 

333 * Decimal formats are generally not synchronized.
334 * It is recommended to create separate format instances for each thread.
335 * If multiple threads access a format concurrently, it must be synchronized
336 * externally.
337 *
338 * 

#### Example


339 *
340 * 

```
{@code
341 * // Print out a number using the localized number, integer, currency,
342 * // and percent format for each locale
343 * Locale[] locales = NumberFormat.getAvailableLocales();
344 * double myNumber = -1234.56;
345 * NumberFormat form;
346 * for (int j = 0; j < 4; ++j) {
347 * System.out.println("FORMAT");
348 * for (int i = 0; i < locales.length; ++i) {
349 * if (locales[i].getCountry().length() == 0) {
350 * continue; // Skip language-only locales
351 * }
352 * System.out.print(locales[i].getDisplayName());
353 * switch (j) {
```


```

```

354 *         case 0:
355 *             form = NumberFormat.getInstance(locales[i]); break;
356 *         case 1:
357 *             form = NumberFormat.getIntegerInstance(locales[i]); break;
358 *         case 2:
359 *             form = NumberFormat.getCurrencyInstance(locales[i]); break;
360 *         default:
361 *             form = NumberFormat.getPercentInstance(locales[i]); break;
362 *     }
363 *     if (form instanceof DecimalFormat) {
364 *         System.out.print(": " + ((DecimalFormat) form).toPattern());
365 *     }
366 *     System.out.print(" -> " + form.format(myNumber));
367 *     try {
368 *         System.out.println(" -> " + form.parse(form.format(myNumber)));
369 *     } catch (ParseException e) {}
370 * }
371 * }
372 * }</pre></blockquote>
373 *
374 * @see      <a href="https://docs.oracle.com/javase/tutorial/i18n/format/decimalFormat.html">
           Java Tutorial</a>
375 * @see      NumberFormat
376 * @see      DecimalFormatSymbols
377 * @see      ParsePosition
378 * @author    Mark Davis
379 * @author    Alan Liu
380 */
381 public class DecimalFormat extends NumberFormat {
382
383     /**
384      * Creates a DecimalFormat using the default pattern and symbols
385      * for the default {@link java.util.Locale.Category#FORMAT FORMAT} locale.
386      * This is a convenient way to obtain a
387      * DecimalFormat when internationalization is not the main concern.
388      * <p>
389      * To obtain standard formats for a given locale, use the factory methods
390      * on NumberFormat such as getNumberInstance. These factories will
391      * return the most appropriate sub-class of NumberFormat for a given
392      * locale.
393      *
394      * @see java.text.NumberFormat#getInstance
395      * @see java.text.NumberFormat#getNumberInstance
396      * @see java.text.NumberFormat#getCurrencyInstance
397      * @see java.text.NumberFormat#getPercentInstance
398      */
399     public DecimalFormat() {
400         // Get the pattern for the default locale.
401         Locale def = Locale.getDefault(Locale.Category.FORMAT);
402         LocaleProviderAdapter adapter = LocaleProviderAdapter.getAdapter(NumberFormatProvider.class, def);
403         if (!(adapter instanceof ResourceBundleBasedAdapter)) {
404             adapter = LocaleProviderAdapter.getResourceBundleBased();

```



```

405     }
406     String[] all = adapter.getLocaleResources(def).getNumberPatterns();
407
408     // Always applyPattern after the symbols are set
409     this.symbols = DecimalFormatSymbols.getInstance(def);
410     applyPattern(all[0], false);
411 }
412
413
414 /**
415  * Creates a DecimalFormat using the given pattern and the symbols
416  * for the default {@link java.util.Locale.Category#FORMAT FORMAT} locale.
417  * This is a convenient way to obtain a
418  * DecimalFormat when internationalization is not the main concern.
419  * <p>
420  * To obtain standard formats for a given locale, use the factory methods
421  * on NumberFormat such as getNumberInstance. These factories will
422  * return the most appropriate sub-class of NumberFormat for a given
423  * locale.
424  *
425  * @param pattern a non-localized pattern string.
426  * @exception NullPointerException if <code>pattern</code> is null
427  * @exception IllegalArgumentException if the given pattern is invalid.
428  * @see java.text.NumberFormat#getInstance
429  * @see java.text.NumberFormat#getNumberInstance
430  * @see java.text.NumberFormat#getCurrencyInstance
431  * @see java.text.NumberFormat#getPercentInstance
432  */
433 public DecimalFormat(String pattern) {
434     // Always applyPattern after the symbols are set
435     this.symbols = DecimalFormatSymbols.getInstance(Locale.getDefault(Locale.Category.FORMAT));
436     applyPattern(pattern, false);
437 }
438
439
440 /**
441  * Creates a DecimalFormat using the given pattern and symbols.
442  * Use this constructor when you need to completely customize the
443  * behavior of the format.
444  * <p>
445  * To obtain standard formats for a given
446  * locale, use the factory methods on NumberFormat such as
447  * getInstance or getCurrencyInstance. If you need only minor adjustments
448  * to a standard format, you can modify the format returned by
449  * a NumberFormat factory method.
450  *
451  * @param pattern a non-localized pattern string
452  * @param symbols the set of symbols to be used
453  * @exception NullPointerException if any of the given arguments is null
454  * @exception IllegalArgumentException if the given pattern is invalid
455  * @see java.text.NumberFormat#getInstance
456  * @see java.text.NumberFormat#getNumberInstance
457  * @see java.text.NumberFormat#getCurrencyInstance

```

```

458     * @see java.text.NumberFormat#getPercentInstance
459     * @see java.text.DecimalFormatSymbols
460     */
461     public DecimalFormat (String pattern, DecimalFormatSymbols symbols) {
462         // Always applyPattern after the symbols are set
463         this.symbols = (DecimalFormatSymbols)symbols.clone();
464         applyPattern(pattern, false);
465     }
466
467     // Overrides
468     /**
469     * Formats a number and appends the resulting text to the given string
470     * buffer.
471     * The number can be of any subclass of {@link java.lang.Number}.
472     * <p>
473     * This implementation uses the maximum precision permitted.
474     * @param number    the number to format
475     * @param toAppendTo the <code>StringBuffer</code> to which the formatted
476     *                  text is to be appended
477     * @param pos       On input: an alignment field, if desired.
478     *                  On output: the offsets of the alignment field.
479     * @return          the value passed in as <code>toAppendTo</code>
480     * @exception       IllegalArgumentException if <code>number</code> is
481     *                  null or not an instance of <code>Number</code>.
482     * @exception       NullPointerException if <code>toAppendTo</code> or
483     *                  <code>pos</code> is null
484     * @exception       ArithmeticException if rounding is needed with rounding
485     *                  mode being set to RoundingMode.UNNECESSARY
486     * @see             java.text.FieldPosition
487     */
488     @Override
489     public final StringBuffer format(Object number,
490                                     StringBuffer toAppendTo,
491                                     FieldPosition pos) {
492         if (number instanceof Long || number instanceof Integer ||
493             number instanceof Short || number instanceof Byte ||
494             number instanceof AtomicInteger ||
495             number instanceof AtomicLong ||
496             (number instanceof BigInteger &&
497              ((BigInteger)number).bitLength () < 64)) {
498             return format(((Number)number).longValue(), toAppendTo, pos);
499         } else if (number instanceof BigDecimal) {
500             return format((BigDecimal)number, toAppendTo, pos);
501         } else if (number instanceof BigInteger) {
502             return format((BigInteger)number, toAppendTo, pos);
503         } else if (number instanceof Number) {
504             return format(((Number)number).doubleValue(), toAppendTo, pos);
505         } else {
506             throw new IllegalArgumentException("Cannot format given Object as a Number");
507         }
508     }
509 }
510

```

```

511 /**
512  * Formats a double to produce a string.
513  * @param number    The double to format
514  * @param result     where the text is to be appended
515  * @param fieldPosition  On input: an alignment field, if desired.
516  * On output: the offsets of the alignment field.
517  * @exception ArithmeticException if rounding is needed with rounding
518  * mode being set to RoundingMode.UNNECESSARY
519  * @return The formatted number string
520  * @see java.text.FieldPosition
521  */
522 @Override
523 public StringBuffer format(double number, StringBuffer result,
524                           FieldPosition fieldPosition) {
525     // If fieldPosition is a DontCareFieldPosition instance we can
526     // try to go to fast-path code.
527     boolean tryFastPath = false;
528     if (fieldPosition == DontCareFieldPosition.INSTANCE)
529         tryFastPath = true;
530     else {
531         fieldPosition.setBeginIndex(0);
532         fieldPosition.setEndIndex(0);
533     }
534
535     if (tryFastPath) {
536         String tempResult = fastFormat(number);
537         if (tempResult != null) {
538             result.append(tempResult);
539             return result;
540         }
541     }
542
543     // if fast-path could not work, we fallback to standard code.
544     return format(number, result, fieldPosition.getFieldDelegate());
545 }
546
547 /**
548  * Formats a double to produce a string.
549  * @param number    The double to format
550  * @param result     where the text is to be appended
551  * @param delegate notified of locations of sub fields
552  * @exception ArithmeticException if rounding is needed with rounding
553  * mode being set to RoundingMode.UNNECESSARY
554  * @return The formatted number string
555  */
556 private StringBuffer format(double number, StringBuffer result,
557                             FieldDelegate delegate) {
558     if (Double.isNaN(number) ||
559         (Double.isInfinite(number) && multiplier == 0)) {
560         int iFieldStart = result.length();
561         result.append(symbols.getNaN());
562         delegate.formatted(INTEGER_FIELD, Field.INTEGER, Field.INTEGER,
563                           iFieldStart, result.length(), result);

```

```

564         return result;
565     }
566
567     /* Detecting whether a double is negative is easy with the exception of
568     * the value -0.0. This is a double which has a zero mantissa (and
569     * exponent), but a negative sign bit. It is semantically distinct from
570     * a zero with a positive sign bit, and this distinction is important
571     * to certain kinds of computations. However, it's a little tricky to
572     * detect, since (-0.0 == 0.0) and !(-0.0 < 0.0). How then, you may
573     * ask, does it behave distinctly from +0.0? Well, 1/(-0.0) ==
574     * -Infinity. Proper detection of -0.0 is needed to deal with the
575     * issues raised by bugs 4106658, 4106667, and 4147706. Liu 7/6/98.
576     */
577     boolean isNegative = ((number < 0.0) || (number == 0.0 && 1/number < 0.0)) ^ (multiplier <
        0);
578
579     if (multiplier != 1) {
580         number *= multiplier;
581     }
582
583     if (Double.isInfinite(number)) {
584         if (isNegative) {
585             append(result, negativePrefix, delegate,
586                 getNegativePrefixFieldPositions(), Field.SIGN);
587         } else {
588             append(result, positivePrefix, delegate,
589                 getPositivePrefixFieldPositions(), Field.SIGN);
590         }
591
592         int iFieldStart = result.length();
593         result.append(symbols.getInfinity());
594         delegate.formatted(INTEGER_FIELD, Field.INTEGER, Field.INTEGER,
595             iFieldStart, result.length(), result);
596
597         if (isNegative) {
598             append(result, negativeSuffix, delegate,
599                 getNegativeSuffixFieldPositions(), Field.SIGN);
600         } else {
601             append(result, positiveSuffix, delegate,
602                 getPositiveSuffixFieldPositions(), Field.SIGN);
603         }
604
605         return result;
606     }
607
608     if (isNegative) {
609         number = -number;
610     }
611
612     // at this point we are guaranteed a nonnegative finite number.
613     assert(number >= 0 && !Double.isInfinite(number));
614
615     synchronized(digitList) {

```

```

616         int maxIntDigits = super.getMaximumIntegerDigits();
617         int minIntDigits = super.getMinimumIntegerDigits();
618         int maxFraDigits = super.getMaximumFractionDigits();
619         int minFraDigits = super.getMinimumFractionDigits();
620
621         digitList.set(isNegative, number, useExponentialNotation ?
622             maxIntDigits + maxFraDigits : maxFraDigits,
623             !useExponentialNotation);
624         return subformat(result, delegate, isNegative, false,
625             maxIntDigits, minIntDigits, maxFraDigits, minFraDigits);
626     }
627 }
628
629 /**
630  * Format a long to produce a string.
631  * @param number    The long to format
632  * @param result    where the text is to be appended
633  * @param fieldPosition    On input: an alignment field, if desired.
634  * On output: the offsets of the alignment field.
635  * @exception    ArithmeticException if rounding is needed with rounding
636  * mode being set to RoundingMode.UNNECESSARY
637  * @return The formatted number string
638  * @see java.text.FieldPosition
639  */
640 @Override
641 public StringBuffer format(long number, StringBuffer result,
642     FieldPosition fieldPosition) {
643     fieldPosition.setBeginIndex(0);
644     fieldPosition.setEndIndex(0);
645
646     return format(number, result, fieldPosition.getFieldDelegate());
647 }
648
649 /**
650  * Format a long to produce a string.
651  * @param number    The long to format
652  * @param result    where the text is to be appended
653  * @param delegate notified of locations of sub fields
654  * @return The formatted number string
655  * @exception    ArithmeticException if rounding is needed with rounding
656  * mode being set to RoundingMode.UNNECESSARY
657  * @see java.text.FieldPosition
658  */
659 private StringBuffer format(long number, StringBuffer result,
660     FieldDelegate delegate) {
661     boolean isNegative = (number < 0);
662     if (isNegative) {
663         number = -number;
664     }
665
666     // In general, long values always represent real finite numbers, so
667     // we don't have to check for +/- Infinity or NaN. However, there
668     // is one case we have to be careful of: The multiplier can push

```

```

669 // a number near MIN_VALUE or MAX_VALUE outside the legal range. We
670 // check for this before multiplying, and if it happens we use
671 // BigInteger instead.
672 boolean useBigInteger = false;
673 if (number < 0) { // This can only happen if number == Long.MIN_VALUE.
674     if (multiplier != 0) {
675         useBigInteger = true;
676     }
677 } else if (multiplier != 1 && multiplier != 0) {
678     long cutoff = Long.MAX_VALUE / multiplier;
679     if (cutoff < 0) {
680         cutoff = -cutoff;
681     }
682     useBigInteger = (number > cutoff);
683 }
684
685 if (useBigInteger) {
686     if (isNegative) {
687         number = -number;
688     }
689     BigInteger bigIntegerValue = BigInteger.valueOf(number);
690     return format(bigIntegerValue, result, delegate, true);
691 }
692
693 number *= multiplier;
694 if (number == 0) {
695     isNegative = false;
696 } else {
697     if (multiplier < 0) {
698         number = -number;
699         isNegative = !isNegative;
700     }
701 }
702
703 synchronized(digitList) {
704     int maxIntDigits = super.getMaximumIntegerDigits();
705     int minIntDigits = super.getMinimumIntegerDigits();
706     int maxFraDigits = super.getMaximumFractionDigits();
707     int minFraDigits = super.getMinimumFractionDigits();
708
709     digitList.set(isNegative, number,
710         useExponentialNotation ? maxIntDigits + maxFraDigits : 0);
711
712     return subformat(result, delegate, isNegative, true,
713         maxIntDigits, minIntDigits, maxFraDigits, minFraDigits);
714 }
715 }
716
717 /**
718  * Formats a BigDecimal to produce a string.
719  * @param number    The BigDecimal to format
720  * @param result    where the text is to be appended
721  * @param fieldPosition    On input: an alignment field, if desired.

```

```

722     * On output: the offsets of the alignment field.
723     * @return The formatted number string
724     * @exception      ArithmeticException if rounding is needed with rounding
725     *                  mode being set to RoundingMode.UNNECESSARY
726     * @see java.text.FieldPosition
727     */
728     private StringBuffer format(BigDecimal number, StringBuffer result,
729                               FieldPosition fieldPosition) {
730         fieldPosition.setBeginIndex(0);
731         fieldPosition.setEndIndex(0);
732         return format(number, result, fieldPosition.getFieldDelegate());
733     }
734
735     /**
736     * Formats a BigDecimal to produce a string.
737     * @param number      The BigDecimal to format
738     * @param result       where the text is to be appended
739     * @param delegate notified of locations of sub fields
740     * @exception      ArithmeticException if rounding is needed with rounding
741     *                  mode being set to RoundingMode.UNNECESSARY
742     * @return The formatted number string
743     */
744     private StringBuffer format(BigDecimal number, StringBuffer result,
745                               FieldDelegate delegate) {
746         if (multiplier != 1) {
747             number = number.multiply(getBigDecimalMultiplier());
748         }
749         boolean isNegative = number.signum() == -1;
750         if (isNegative) {
751             number = number.negate();
752         }
753
754         synchronized(digitList) {
755             int maxIntDigits = getMaximumIntegerDigits();
756             int minIntDigits = getMinimumIntegerDigits();
757             int maxFraDigits = getMaximumFractionDigits();
758             int minFraDigits = getMinimumFractionDigits();
759             int maximumDigits = maxIntDigits + maxFraDigits;
760
761             digitList.set(isNegative, number, useExponentialNotation ?
762                          ((maximumDigits < 0) ? Integer.MAX_VALUE : maximumDigits) :
763                          maxFraDigits, !useExponentialNotation);
764
765             return subformat(result, delegate, isNegative, false,
766                             maxIntDigits, minIntDigits, maxFraDigits, minFraDigits);
767         }
768     }
769
770     /**
771     * Format a BigInteger to produce a string.
772     * @param number      The BigInteger to format
773     * @param result       where the text is to be appended
774     * @param fieldPosition On input: an alignment field, if desired.

```

```

775     * On output: the offsets of the alignment field.
776     * @return The formatted number string
777     * @exception      ArithmeticException if rounding is needed with rounding
778     *                  mode being set to RoundingMode.UNNECESSARY
779     * @see java.text.FieldPosition
780     */
781     private StringBuffer format(BigInteger number, StringBuffer result,
782                                FieldPosition fieldPosition) {
783         fieldPosition.setBeginIndex(0);
784         fieldPosition.setEndIndex(0);
785
786         return format(number, result, fieldPosition.getFieldDelegate(), false);
787     }
788
789     /**
790     * Format a BigInteger to produce a string.
791     * @param number      The BigInteger to format
792     * @param result      where the text is to be appended
793     * @param delegate notified of locations of sub fields
794     * @return The formatted number string
795     * @exception      ArithmeticException if rounding is needed with rounding
796     *                  mode being set to RoundingMode.UNNECESSARY
797     * @see java.text.FieldPosition
798     */
799     private StringBuffer format(BigInteger number, StringBuffer result,
800                                FieldDelegate delegate, boolean formatLong) {
801         if (multiplier != 1) {
802             number = number.multiply(getBigIntegerMultiplier());
803         }
804         boolean isNegative = number.signum() == -1;
805         if (isNegative) {
806             number = number.negate();
807         }
808
809         synchronized(digitList) {
810             int maxIntDigits, minIntDigits, maxFraDigits, minFraDigits, maximumDigits;
811             if (formatLong) {
812                 maxIntDigits = super.getMaximumIntegerDigits();
813                 minIntDigits = super.getMinimumIntegerDigits();
814                 maxFraDigits = super.getMaximumFractionDigits();
815                 minFraDigits = super.getMinimumFractionDigits();
816                 maximumDigits = maxIntDigits + maxFraDigits;
817             } else {
818                 maxIntDigits = getMaximumIntegerDigits();
819                 minIntDigits = getMinimumIntegerDigits();
820                 maxFraDigits = getMaximumFractionDigits();
821                 minFraDigits = getMinimumFractionDigits();
822                 maximumDigits = maxIntDigits + maxFraDigits;
823                 if (maximumDigits < 0) {
824                     maximumDigits = Integer.MAX_VALUE;
825                 }
826             }
827

```



```

828         digitList.set(isNegative, number,
829                       useExponentialNotation ? maximumDigits : 0);
830
831         return subformat(result, delegate, isNegative, true,
832                          maxIntDigits, minIntDigits, maxFraDigits, minFraDigits);
833     }
834 }
835
836 /**
837  * Formats an Object producing an AttributedCharacterIterator.
838  * You can use the returned AttributedCharacterIterator
839  * to build the resulting String, as well as to determine information
840  * about the resulting String.
841  * <p>
842  * Each attribute key of the AttributedCharacterIterator will be of type
843  * NumberFormat.Field, with the attribute value being the
844  * same as the attribute key.
845  *
846  * @exception NullPointerException if obj is null.
847  * @exception IllegalArgumentException when the Format cannot format the
848  *         given object.
849  * @exception ArithmeticException if rounding is needed with rounding
850  *         mode being set to RoundingMode.UNNECESSARY
851  * @param obj The object to format
852  * @return AttributedCharacterIterator describing the formatted value.
853  * @since 1.4
854  */
855 @Override
856 public AttributedCharacterIterator formatToCharacterIterator(Object obj) {
857     CharacterIteratorFieldDelegate delegate =
858         new CharacterIteratorFieldDelegate();
859     StringBuffer sb = new StringBuffer();
860
861     if (obj instanceof Double || obj instanceof Float) {
862         format(((Number)obj).doubleValue(), sb, delegate);
863     } else if (obj instanceof Long || obj instanceof Integer ||
864               obj instanceof Short || obj instanceof Byte ||
865               obj instanceof AtomicInteger || obj instanceof AtomicLong) {
866         format(((Number)obj).longValue(), sb, delegate);
867     } else if (obj instanceof BigDecimal) {
868         format((BigDecimal)obj, sb, delegate);
869     } else if (obj instanceof BigInteger) {
870         format((BigInteger)obj, sb, delegate, false);
871     } else if (obj == null) {
872         throw new NullPointerException(
873             "formatToCharacterIterator must be passed non-null object");
874     } else {
875         throw new IllegalArgumentException(
876             "Cannot format given Object as a Number");
877     }
878     return delegate.getIterator(sb.toString());
879 }
880

```

```

881 // ==== Begin fast-path formatting logic for double =====
882
883 /* Fast-path formatting will be used for format(double ...) methods iff a
884  * number of conditions are met (see checkAndSetFastPathStatus()):
885  * - Only if instance properties meet the right predefined conditions.
886  * - The abs value of the double to format is <= Integer.MAX_VALUE.
887  *
888  * The basic approach is to split the binary to decimal conversion of a
889  * double value into two phases:
890  * * The conversion of the integer portion of the double.
891  * * The conversion of the fractional portion of the double
892  *   (limited to two or three digits).
893  *
894  * The isolation and conversion of the integer portion of the double is
895  * straightforward. The conversion of the fraction is more subtle and relies
896  * on some rounding properties of double to the decimal precisions in
897  * question. Using the terminology of BigDecimal, this fast-path algorithm
898  * is applied when a double value has a magnitude less than Integer.MAX_VALUE
899  * and rounding is to nearest even and the destination format has two or
900  * three digits of *scale* (digits after the decimal point).
901  *
902  * Under a rounding to nearest even policy, the returned result is a digit
903  * string of a number in the (in this case decimal) destination format
904  * closest to the exact numerical value of the (in this case binary) input
905  * value. If two destination format numbers are equally distant, the one
906  * with the last digit even is returned. To compute such a correctly rounded
907  * value, some information about digits beyond the smallest returned digit
908  * position needs to be consulted.
909  *
910  * In general, a guard digit, a round digit, and a sticky *bit* are needed
911  * beyond the returned digit position. If the discarded portion of the input
912  * is sufficiently large, the returned digit string is incremented. In round
913  * to nearest even, this threshold to increment occurs near the half-way
914  * point between digits. The sticky bit records if there are any remaining
915  * trailing digits of the exact input value in the new format; the sticky bit
916  * is consulted only in close to half-way rounding cases.
917  *
918  * Given the computation of the digit and bit values, rounding is then
919  * reduced to a table lookup problem. For decimal, the even/odd cases look
920  * like this:
921  *
922  * Last   Round   Sticky
923  * 6       5       0   => 6   // exactly halfway, return even digit.
924  * 6       5       1   => 7   // a little bit more than halfway, round up.
925  * 7       5       0   => 8   // exactly halfway, round up to even.
926  * 7       5       1   => 8   // a little bit more than halfway, round up.
927  * With analogous entries for other even and odd last-returned digits.
928  *
929  * However, decimal negative powers of 5 smaller than 0.5 are *not* exactly
930  * representable as binary fraction. In particular, 0.005 (the round limit
931  * for a two-digit scale) and 0.0005 (the round limit for a three-digit
932  * scale) are not representable. Therefore, for input values near these cases
933  * the sticky bit is known to be set which reduces the rounding logic to:

```

```

934      *
935      * Last   Round   Sticky
936      * 6     5       1     => 7   // a little bit more than halfway, round up.
937      * 7     5       1     => 8   // a little bit more than halfway, round up.
938      *
939      * In other words, if the round digit is 5, the sticky bit is known to be
940      * set. If the round digit is something other than 5, the sticky bit is not
941      * relevant. Therefore, some of the logic about whether or not to increment
942      * the destination *decimal* value can occur based on tests of *binary*
943      * computations of the binary input number.
944      */
945
946  /**
947   * Check validity of using fast-path for this instance. If fast-path is valid
948   * for this instance, sets fast-path state as true and initializes fast-path
949   * utility fields as needed.
950   *
951   * This method is supposed to be called rarely, otherwise that will break the
952   * fast-path performance. That means avoiding frequent changes of the
953   * properties of the instance, since for most properties, each time a change
954   * happens, a call to this method is needed at the next format call.
955   *
956   * FAST-PATH RULES:
957   * Similar to the default DecimalFormat instantiation case.
958   * More precisely:
959   * - HALF_EVEN rounding mode,
960   * - isGroupingUsed() is true,
961   * - groupingSize of 3,
962   * - multiplier is 1,
963   * - Decimal separator not mandatory,
964   * - No use of exponential notation,
965   * - minimumIntegerDigits is exactly 1 and maximumIntegerDigits at least 10
966   * - For number of fractional digits, the exact values found in the default case:
967   *   Currency : min = max = 2.
968   *   Decimal  : min = 0. max = 3.
969   *
970   */
971  private void checkAndSetFastPathStatus() {
972
973      boolean fastPathWasOn = isFastPath;
974
975      if ((roundingMode == RoundingMode.HALF_EVEN) &&
976          (isGroupingUsed()) &&
977          (groupingSize == 3) &&
978          (multiplier == 1) &&
979          (!decimalSeparatorAlwaysShown) &&
980          (!useExponentialNotation)) {
981
982          // The fast-path algorithm is semi-hardcoded against
983          // minimumIntegerDigits and maximumIntegerDigits.
984          isFastPath = ((minimumIntegerDigits == 1) &&
985                      (maximumIntegerDigits >= 10));
986

```

```

987     // The fast-path algorithm is hardcoded against
988     // minimumFractionDigits and maximumFractionDigits.
989     if (isFastPath) {
990         if (isCurrencyFormat) {
991             if ((minimumFractionDigits != 2) ||
992                 (maximumFractionDigits != 2))
993                 isFastPath = false;
994         } else if ((minimumFractionDigits != 0) ||
995                     (maximumFractionDigits != 3))
996             isFastPath = false;
997     }
998 } else
999     isFastPath = false;
1000
1001 // Since some instance properties may have changed while still falling
1002 // in the fast-path case, we need to reinitialize fastPathData anyway.
1003 if (isFastPath) {
1004     // We need to instantiate fastPathData if not already done.
1005     if (fastPathData == null)
1006         fastPathData = new FastPathData();
1007
1008     // Sets up the locale specific constants used when formatting.
1009     // '0' is our default representation of zero.
1010     fastPathData.zeroDelta = symbols.getZeroDigit() - '0';
1011     fastPathData.groupingChar = symbols.getGroupingSeparator();
1012
1013     // Sets up fractional constants related to currency/decimal pattern.
1014     fastPathData.fractionalMaxIntBound = (isCurrencyFormat) ? 99 : 999;
1015     fastPathData.fractionalScaleFactor = (isCurrencyFormat) ? 100.0d : 1000.0d;
1016
1017     // Records the need for adding prefix or suffix
1018     fastPathData.positiveAffixesRequired =
1019         (positivePrefix.length() != 0) || (positiveSuffix.length() != 0);
1020     fastPathData.negativeAffixesRequired =
1021         (negativePrefix.length() != 0) || (negativeSuffix.length() != 0);
1022
1023     // Creates a cached char container for result, with max possible size.
1024     int maxNbIntegralDigits = 10;
1025     int maxNbGroups = 3;
1026     int containerSize =
1027         Math.max(positivePrefix.length(), negativePrefix.length()) +
1028         maxNbIntegralDigits + maxNbGroups + 1 + maximumFractionDigits +
1029         Math.max(positiveSuffix.length(), negativeSuffix.length());
1030
1031     fastPathData.fastPathContainer = new char[containerSize];
1032
1033     // Sets up prefix and suffix char arrays constants.
1034     fastPathData.charsPositiveSuffix = positiveSuffix.toCharArray();
1035     fastPathData.charsNegativeSuffix = negativeSuffix.toCharArray();
1036     fastPathData.charsPositivePrefix = positivePrefix.toCharArray();
1037     fastPathData.charsNegativePrefix = negativePrefix.toCharArray();
1038
1039     // Sets up fixed index positions for integral and fractional digits.

```

```

1040         // Sets up decimal point in cached result container.
1041         int longestPrefixLength =
1042             Math.max(positivePrefix.length(), negativePrefix.length());
1043         int decimalPointIndex =
1044             maxNbIntegralDigits + maxNbGroups + longestPrefixLength;
1045
1046         fastPathData.integralLastIndex = decimalPointIndex - 1;
1047         fastPathData.fractionalFirstIndex = decimalPointIndex + 1;
1048         fastPathData.fastPathContainer[decimalPointIndex] =
1049             isCurrencyFormat ?
1050                 symbols.getMonetaryDecimalSeparator() :
1051                 symbols.getDecimalSeparator();
1052
1053     } else if (fastPathWasOn) {
1054         // Previous state was fast-path and is no more.
1055         // Resets cached array constants.
1056         fastPathData.fastPathContainer = null;
1057         fastPathData.charsPositiveSuffix = null;
1058         fastPathData.charsNegativeSuffix = null;
1059         fastPathData.charsPositivePrefix = null;
1060         fastPathData.charsNegativePrefix = null;
1061     }
1062
1063     fastPathCheckNeeded = false;
1064 }
1065
1066 /**
1067  * Returns true if rounding-up must be done on {@code scaledFractionalPartAsInt},
1068  * false otherwise.
1069  *
1070  * This is a utility method that takes correct half-even rounding decision on
1071  * passed fractional value at the scaled decimal point (2 digits for currency
1072  * case and 3 for decimal case), when the approximated fractional part after
1073  * scaled decimal point is exactly 0.5d. This is done by means of exact
1074  * calculations on the {@code fractionalPart} floating-point value.
1075  *
1076  * This method is supposed to be called by private {@code fastDoubleFormat}
1077  * method only.
1078  *
1079  * The algorithms used for the exact calculations are :
1080  *
1081  * The FastTwoSum algorithm, from T.J.Dekker, described in the
1082  * papers "A Floating-Point Technique for Extending the Available
1083  * Precision" by Dekker, and in "Adaptive Precision Floating-Point
1084  * Arithmetic and Fast Robust Geometric Predicates" from J.Shewchuk.
1085  *
1086  * A modified version of Sum2S cascaded summation described in
1087  * "Accurate Sum and Dot Product" from Takeshi Ogita and All. As
1088  * Ogita says in this paper this is an equivalent of the Kahan-Babuska's
1089  * summation algorithm because we order the terms by magnitude before summing
1090  * them. For this reason we can use the FastTwoSum algorithm rather
1091  * than the more expensive Knuth's TwoSum.
1092  *

```

```

1093 * We do this to avoid a more expensive exact "<i>TwoProduct</i>" algorithm,
1094 * like those described in Shewchuk's paper above. See comments in the code
1095 * below.
1096 *
1097 * @param fractionalPart The fractional value on which we take rounding
1098 * decision.
1099 * @param scaledFractionalPartAsInt The integral part of the scaled
1100 * fractional value.
1101 *
1102 * @return the decision that must be taken regarding half-even rounding.
1103 */
1104 private boolean exactRoundUp(double fractionalPart,
1105                             int scaledFractionalPartAsInt) {
1106
1107     /* exactRoundUp() method is called by fastDoubleFormat() only.
1108     * The precondition expected to be verified by the passed parameters is :
1109     * scaledFractionalPartAsInt ==
1110     *     (int) (fractionalPart * fastPathData.fractionalScaleFactor).
1111     * This is ensured by fastDoubleFormat() code.
1112     */
1113
1114     /* We first calculate roundoff error made by fastDoubleFormat() on
1115     * the scaled fractional part. We do this with exact calculation on the
1116     * passed fractionalPart. Rounding decision will then be taken from roundoff.
1117     */
1118
1119     /* ---- TwoProduct(fractionalPart, scale factor (i.e. 1000.0d or 100.0d)).
1120     *
1121     * The below is an optimized exact "TwoProduct" calculation of passed
1122     * fractional part with scale factor, using Ogita's Sum2S cascaded
1123     * summation adapted as Kahan-Babuska equivalent by using FastTwoSum
1124     * (much faster) rather than Knuth's TwoSum.
1125     *
1126     * We can do this because we order the summation from smallest to
1127     * greatest, so that FastTwoSum can be used without any additional error.
1128     *
1129     * The "TwoProduct" exact calculation needs 17 flops. We replace this by
1130     * a cascaded summation of FastTwoSum calculations, each involving an
1131     * exact multiply by a power of 2.
1132     *
1133     * Doing so saves overall 4 multiplications and 1 addition compared to
1134     * using traditional "TwoProduct".
1135     *
1136     * The scale factor is either 100 (currency case) or 1000 (decimal case).
1137     * - when 1000, we replace it by (1024 - 16 - 8) = 1000.
1138     * - when 100, we replace it by (128 - 32 + 4) = 100.
1139     * Every multiplication by a power of 2 (1024, 128, 32, 16, 8, 4) is exact.
1140     *
1141     */
1142     double approxMax;    // Will always be positive.
1143     double approxMedium; // Will always be negative.
1144     double approxMin;
1145

```

```

1146     double fastTwoSumApproximation = 0.0d;
1147     double fastTwoSumRoundOff = 0.0d;
1148     double bVirtual = 0.0d;
1149
1150     if (isCurrencyFormat) {
1151         // Scale is 100 = 128 - 32 + 4.
1152         // Multiply by 2**n is a shift. No roundoff. No error.
1153         approxMax    = fractionalPart * 128.00d;
1154         approxMedium = - (fractionalPart * 32.00d);
1155         approxMin    = fractionalPart * 4.00d;
1156     } else {
1157         // Scale is 1000 = 1024 - 16 - 8.
1158         // Multiply by 2**n is a shift. No roundoff. No error.
1159         approxMax    = fractionalPart * 1024.00d;
1160         approxMedium = - (fractionalPart * 16.00d);
1161         approxMin    = - (fractionalPart * 8.00d);
1162     }
1163
1164     // Shewchuk/Dekker's FastTwoSum(approxMedium, approxMin).
1165     assert(-approxMedium >= Math.abs(approxMin));
1166     fastTwoSumApproximation = approxMedium + approxMin;
1167     bVirtual = fastTwoSumApproximation - approxMedium;
1168     fastTwoSumRoundOff = approxMin - bVirtual;
1169     double approxS1 = fastTwoSumApproximation;
1170     double roundoffS1 = fastTwoSumRoundOff;
1171
1172     // Shewchuk/Dekker's FastTwoSum(approxMax, approxS1);
1173     assert(approxMax >= Math.abs(approxS1));
1174     fastTwoSumApproximation = approxMax + approxS1;
1175     bVirtual = fastTwoSumApproximation - approxMax;
1176     fastTwoSumRoundOff = approxS1 - bVirtual;
1177     double roundoff1000 = fastTwoSumRoundOff;
1178     double approx1000 = fastTwoSumApproximation;
1179     double roundoffTotal = roundoffS1 + roundoff1000;
1180
1181     // Shewchuk/Dekker's FastTwoSum(approx1000, roundoffTotal);
1182     assert(approx1000 >= Math.abs(roundoffTotal));
1183     fastTwoSumApproximation = approx1000 + roundoffTotal;
1184     bVirtual = fastTwoSumApproximation - approx1000;
1185
1186     // Now we have got the roundoff for the scaled fractional
1187     double scaledFractionalRoundoff = roundoffTotal - bVirtual;
1188
1189     // ---- TwoProduct(fractionalPart, scale (i.e. 1000.0d or 100.0d)) end.
1190
1191     /* ---- Taking the rounding decision
1192     *
1193     * We take rounding decision based on roundoff and half-even rounding
1194     * rule.
1195     *
1196     * The above TwoProduct gives us the exact roundoff on the approximated
1197     * scaled fractional, and we know that this approximation is exactly
1198     * 0.5d, since that has already been tested by the caller

```



```

1199     * (fastDoubleFormat).
1200     *
1201     * Decision comes first from the sign of the calculated exact roundoff.
1202     * - Since being exact roundoff, it cannot be positive with a scaled
1203     *   fractional less than 0.5d, as well as negative with a scaled
1204     *   fractional greater than 0.5d. That leaves us with following 3 cases.
1205     * - positive, thus scaled fractional == 0.500...0fff ==> round-up.
1206     * - negative, thus scaled fractional == 0.499...9fff ==> don't round-up.
1207     * - is zero, thus scaled fractional == 0.5 ==> half-even rounding applies :
1208     *   we round-up only if the integral part of the scaled fractional is odd.
1209     *
1210     */
1211     if (scaledFractionalRoundoff > 0.0) {
1212         return true;
1213     } else if (scaledFractionalRoundoff < 0.0) {
1214         return false;
1215     } else if ((scaledFractionalPartAsInt & 1) != 0) {
1216         return true;
1217     }
1218
1219     return false;
1220
1221     // ---- Taking the rounding decision end
1222 }
1223
1224 /**
1225  * Collects integral digits from passed {@code number}, while setting
1226  * grouping chars as needed. Updates {@code firstUsedIndex} accordingly.
1227  *
1228  * Loops downward starting from {@code backwardIndex} position (inclusive).
1229  *
1230  * @param number The int value from which we collect digits.
1231  * @param digitsBuffer The char array container where digits and grouping chars
1232  *   are stored.
1233  * @param backwardIndex the position from which we start storing digits in
1234  *   digitsBuffer.
1235  *
1236  */
1237 private void collectIntegralDigits(int number,
1238                                   char[] digitsBuffer,
1239                                   int backwardIndex) {
1240     int index = backwardIndex;
1241     int q;
1242     int r;
1243     while (number > 999) {
1244         // Generates 3 digits per iteration.
1245         q = number / 1000;
1246         r = number - (q << 10) + (q << 4) + (q << 3); // -1024 +16 +8 = 1000.
1247         number = q;
1248
1249         digitsBuffer[index--] = DigitArrays.DigitOnes1000[r];
1250         digitsBuffer[index--] = DigitArrays.DigitTens1000[r];
1251         digitsBuffer[index--] = DigitArrays.DigitHundreds1000[r];

```



```

1252     digitsBuffer[index--] = fastPathData.groupingChar;
1253 }
1254
1255     // Collects last 3 or less digits.
1256     digitsBuffer[index] = DigitArrays.DigitOnes1000[number];
1257     if (number > 9) {
1258         digitsBuffer[--index] = DigitArrays.DigitTens1000[number];
1259         if (number > 99)
1260             digitsBuffer[--index] = DigitArrays.DigitHundreds1000[number];
1261     }
1262
1263     fastPathData.firstUsedIndex = index;
1264 }
1265
1266 /**
1267  * Collects the 2 (currency) or 3 (decimal) fractional digits from passed
1268  * {@code number}, starting at {@code startIndex} position
1269  * inclusive. There is no punctuation to set here (no grouping chars).
1270  * Updates {@code fastPathData.lastFreeIndex} accordingly.
1271  *
1272  *
1273  * @param number The int value from which we collect digits.
1274  * @param digitsBuffer The char array container where digits are stored.
1275  * @param startIndex the position from which we start storing digits in
1276  * digitsBuffer.
1277  *
1278  */
1279 private void collectFractionalDigits(int number,
1280                                     char[] digitsBuffer,
1281                                     int startIndex) {
1282     int index = startIndex;
1283
1284     char digitOnes = DigitArrays.DigitOnes1000[number];
1285     char digitTens = DigitArrays.DigitTens1000[number];
1286
1287     if (isCurrencyFormat) {
1288         // Currency case. Always collects fractional digits.
1289         digitsBuffer[index++] = digitTens;
1290         digitsBuffer[index++] = digitOnes;
1291     } else if (number != 0) {
1292         // Decimal case. Hundreds will always be collected
1293         digitsBuffer[index++] = DigitArrays.DigitHundreds1000[number];
1294
1295         // Ending zeros won't be collected.
1296         if (digitOnes != '0') {
1297             digitsBuffer[index++] = digitTens;
1298             digitsBuffer[index++] = digitOnes;
1299         } else if (digitTens != '0')
1300             digitsBuffer[index++] = digitTens;
1301
1302     } else
1303         // This is decimal pattern and fractional part is zero.
1304         // We must remove decimal point from result.

```

```

1305         index--;
1306
1307         fastPathData.lastFreeIndex = index;
1308     }
1309
1310     /**
1311     * Internal utility.
1312     * Adds the passed {@code prefix} and {@code suffix} to {@code container}.
1313     *
1314     * @param container Char array container which to prepend/append the
1315     *   prefix/suffix.
1316     * @param prefix Char sequence to prepend as a prefix.
1317     * @param suffix Char sequence to append as a suffix.
1318     *
1319     */
1320     // private void addAffixes(boolean isNegative, char[] container) {
1321     private void addAffixes(char[] container, char[] prefix, char[] suffix) {
1322
1323         // We add affixes only if needed (affix length > 0).
1324         int pl = prefix.length;
1325         int sl = suffix.length;
1326         if (pl != 0) prependPrefix(prefix, pl, container);
1327         if (sl != 0) appendSuffix(suffix, sl, container);
1328
1329     }
1330
1331     /**
1332     * Prepends the passed {@code prefix} chars to given result
1333     * {@code container}. Updates {@code fastPathData.firstUsedIndex}
1334     * accordingly.
1335     *
1336     * @param prefix The prefix characters to prepend to result.
1337     * @param len The number of chars to prepend.
1338     * @param container Char array container which to prepend the prefix
1339     */
1340     private void prependPrefix(char[] prefix,
1341                               int len,
1342                               char[] container) {
1343
1344         fastPathData.firstUsedIndex -= len;
1345         int startIndex = fastPathData.firstUsedIndex;
1346
1347         // If prefix to prepend is only 1 char long, just assigns this char.
1348         // If prefix is less or equal 4, we use a dedicated algorithm that
1349         // has shown to run faster than System.arraycopy.
1350         // If more than 4, we use System.arraycopy.
1351         if (len == 1)
1352             container[startIndex] = prefix[0];
1353         else if (len <= 4) {
1354             int dstLower = startIndex;
1355             int dstUpper = dstLower + len - 1;
1356             int srcUpper = len - 1;
1357             container[dstLower] = prefix[0];

```

```

1358         container[dstUpper] = prefix[srcUpper];
1359
1360         if (len > 2)
1361             container[++dstLower] = prefix[1];
1362         if (len == 4)
1363             container[--dstUpper] = prefix[2];
1364     } else
1365         System.arraycopy(prefix, 0, container, startIndex, len);
1366 }
1367
1368 /**
1369  * Appends the passed {@code suffix} chars to given result
1370  * {@code container}. Updates {@code fastPathData.lastFreeIndex}
1371  * accordingly.
1372  *
1373  * @param suffix The suffix characters to append to result.
1374  * @param len The number of chars to append.
1375  * @param container Char array container which to append the suffix
1376  */
1377 private void appendSuffix(char[] suffix,
1378                          int len,
1379                          char[] container) {
1380
1381     int startIndex = fastPathData.lastFreeIndex;
1382
1383     // If suffix to append is only 1 char long, just assigns this char.
1384     // If suffix is less or equal 4, we use a dedicated algorithm that
1385     // has shown to run faster than System.arraycopy.
1386     // If more than 4, we use System.arraycopy.
1387     if (len == 1)
1388         container[startIndex] = suffix[0];
1389     else if (len <= 4) {
1390         int dstLower = startIndex;
1391         int dstUpper = dstLower + len - 1;
1392         int srcUpper = len - 1;
1393         container[dstLower] = suffix[0];
1394         container[dstUpper] = suffix[srcUpper];
1395
1396         if (len > 2)
1397             container[++dstLower] = suffix[1];
1398         if (len == 4)
1399             container[--dstUpper] = suffix[2];
1400     } else
1401         System.arraycopy(suffix, 0, container, startIndex, len);
1402
1403     fastPathData.lastFreeIndex += len;
1404 }
1405
1406 /**
1407  * Converts digit chars from {@code digitsBuffer} to current locale.
1408  *
1409  * Must be called before adding affixes since we refer to
1410  * {@code fastPathData.firstUsedIndex} and {@code fastPathData.lastFreeIndex},

```

```

1411     * and do not support affixes (for speed reason).
1412     *
1413     * We loop backward starting from last used index in {@code fastPathData}.
1414     *
1415     * @param digitsBuffer The char array container where the digits are stored.
1416     */
1417     private void localizeDigits(char[] digitsBuffer) {
1418
1419         // We will localize only the digits, using the groupingSize,
1420         // and taking into account fractional part.
1421
1422         // First take into account fractional part.
1423         int digitsCounter =
1424             fastPathData.lastFreeIndex - fastPathData.fractionalFirstIndex;
1425
1426         // The case when there is no fractional digits.
1427         if (digitsCounter < 0)
1428             digitsCounter = groupingSize;
1429
1430         // Only the digits remains to localize.
1431         for (int cursor = fastPathData.lastFreeIndex - 1;
1432             cursor >= fastPathData.firstUsedIndex;
1433             cursor--) {
1434             if (digitsCounter != 0) {
1435                 // This is a digit char, we must localize it.
1436                 digitsBuffer[cursor] += fastPathData.zeroDelta;
1437                 digitsCounter--;
1438             } else {
1439                 // Decimal separator or grouping char. Reinit counter only.
1440                 digitsCounter = groupingSize;
1441             }
1442         }
1443     }
1444
1445     /**
1446     * This is the main entry point for the fast-path format algorithm.
1447     *
1448     * At this point we are sure to be in the expected conditions to run it.
1449     * This algorithm builds the formatted result and puts it in the dedicated
1450     * {@code fastPathData.fastPathContainer}.
1451     *
1452     * @param d the double value to be formatted.
1453     * @param negative Flag precising if {@code d} is negative.
1454     */
1455     private void fastDoubleFormat(double d,
1456                                   boolean negative) {
1457
1458         char[] container = fastPathData.fastPathContainer;
1459
1460         /*
1461         * The principle of the algorithm is to :
1462         * - Break the passed double into its integral and fractional parts
1463         *   converted into integers.

```

```
1464     * - Then decide if rounding up must be applied or not by following
1465     *   the half-even rounding rule, first using approximated scaled
1466     *   fractional part.
1467     * - For the difficult cases (approximated scaled fractional part
1468     *   being exactly 0.5d), we refine the rounding decision by calling
1469     *   exactRoundUp utility method that both calculates the exact roundoff
1470     *   on the approximation and takes correct rounding decision.
1471     * - We round-up the fractional part if needed, possibly propagating the
1472     *   rounding to integral part if we meet a "all-nine" case for the
1473     *   scaled fractional part.
1474     * - We then collect digits from the resulting integral and fractional
1475     *   parts, also setting the required grouping chars on the fly.
1476     * - Then we localize the collected digits if needed, and
1477     * - Finally prepend/append prefix/suffix if any is needed.
1478     */
1479
1480     // Exact integral part of d.
1481     int integralPartAsInt = (int) d;
1482
1483     // Exact fractional part of d (since we subtract it's integral part).
1484     double exactFractionalPart = d - (double) integralPartAsInt;
1485
1486     // Approximated scaled fractional part of d (due to multiplication).
1487     double scaledFractional =
1488         exactFractionalPart * fastPathData.fractionalScaleFactor;
1489
1490     // Exact integral part of scaled fractional above.
1491     int fractionalPartAsInt = (int) scaledFractional;
1492
1493     // Exact fractional part of scaled fractional above.
1494     scaledFractional = scaledFractional - (double) fractionalPartAsInt;
1495
1496     // Only when scaledFractional is exactly 0.5d do we have to do exact
1497     // calculations and take fine-grained rounding decision, since
1498     // approximated results above may lead to incorrect decision.
1499     // Otherwise comparing against 0.5d (strictly greater or less) is ok.
1500     boolean roundItUp = false;
1501     if (scaledFractional >= 0.5d) {
1502         if (scaledFractional == 0.5d) {
1503             // Rounding need fine-grained decision.
1504             roundItUp = exactRoundUp(exactFractionalPart, fractionalPartAsInt);
1505         } else
1506             roundItUp = true;
1507
1508         if (roundItUp) {
1509             // Rounds up both fractional part (and also integral if needed).
1510             if (fractionalPartAsInt < fastPathData.fractionalMaxIntBound) {
1511                 fractionalPartAsInt++;
1512             } else {
1513                 // Propagates rounding to integral part since "all nines" case.
1514                 fractionalPartAsInt = 0;
1515                 integralPartAsInt++;
1516             }
1517         }
1518     }
```

```

1517     }
1518 }
1519
1520 // Collecting digits.
1521 collectFractionalDigits(fractionalPartAsInt, container,
1522                        fastPathData.fractionalFirstIndex);
1523 collectIntegralDigits(integralPartAsInt, container,
1524                      fastPathData.integralLastIndex);
1525
1526 // Localizing digits.
1527 if (fastPathData.zeroDelta != 0)
1528     localizeDigits(container);
1529
1530 // Adding prefix and suffix.
1531 if (negative) {
1532     if (fastPathData.negativeAffixesRequired)
1533         addAffixes(container,
1534                   fastPathData.charsNegativePrefix,
1535                   fastPathData.charsNegativeSuffix);
1536 } else if (fastPathData.positiveAffixesRequired)
1537     addAffixes(container,
1538               fastPathData.charsPositivePrefix,
1539               fastPathData.charsPositiveSuffix);
1540 }
1541
1542 /**
1543  * A fast-path shortcut of format(double) to be called by NumberFormat, or by
1544  * format(double, ...) public methods.
1545  *
1546  * If instance can be applied fast-path and passed double is not NaN or
1547  * Infinity, is in the integer range, we call {@code fastDoubleFormat}
1548  * after changing {@code d} to its positive value if necessary.
1549  *
1550  * Otherwise returns null by convention since fast-path can't be exercised.
1551  *
1552  * @param d The double value to be formatted
1553  *
1554  * @return the formatted result for {@code d} as a string.
1555  */
1556 String fastFormat(double d) {
1557     // (Re-)Evaluates fast-path status if needed.
1558     if (fastPathCheckNeeded)
1559         checkAndSetFastPathStatus();
1560
1561     if (!isFastPath)
1562         // DecimalFormat instance is not in a fast-path state.
1563         return null;
1564
1565     if (!Double.isFinite(d))
1566         // Should not use fast-path for Infinity and NaN.
1567         return null;
1568
1569     // Extracts and records sign of double value, possibly changing it

```

```

1570     // to a positive one, before calling fastDoubleFormat().
1571     boolean negative = false;
1572     if (d < 0.0d) {
1573         negative = true;
1574         d = -d;
1575     } else if (d == 0.0d) {
1576         negative = (Math.copySign(1.0d, d) == -1.0d);
1577         d = +0.0d;
1578     }
1579
1580     if (d > MAX_INT_AS_DOUBLE)
1581         // Filters out values that are outside expected fast-path range
1582         return null;
1583     else
1584         fastDoubleFormat(d, negative);
1585
1586     // Returns a new string from updated fastPathContainer.
1587     return new String(fastPathData.fastPathContainer,
1588                     fastPathData.firstUsedIndex,
1589                     fastPathData.lastFreeIndex - fastPathData.firstUsedIndex);
1590
1591 }
1592
1593 // ===== End fast-path forming logic for double =====
1594
1595 /**
1596  * Complete the formatting of a finite number. On entry, the digitList must
1597  * be filled in with the correct digits.
1598  */
1599 private StringBuffer subformat(StringBuffer result, FieldDelegate delegate,
1600                               boolean isNegative, boolean isInteger,
1601                               int maxIntDigits, int minIntDigits,
1602                               int maxFraDigits, int minFraDigits) {
1603     // NOTE: This isn't required anymore because DigitList takes care of this.
1604     //
1605     // // The negative of the exponent represents the number of leading
1606     // // zeros between the decimal and the first non-zero digit, for
1607     // // a value < 0.1 (e.g., for 0.00123, -fExponent == 2). If this
1608     // // is more than the maximum fraction digits, then we have an underflow
1609     // // for the printed representation. We recognize this here and set
1610     // // the DigitList representation to zero in this situation.
1611     //
1612     // if (-digitList.decimalAt >= getMaximumFractionDigits())
1613     // {
1614     //     digitList.count = 0;
1615     // }
1616
1617     char zero = symbols.getZeroDigit();
1618     int zeroDelta = zero - '0'; // '0' is the DigitList representation of zero
1619     char grouping = symbols.getGroupingSeparator();
1620     char decimal = isCurrencyFormat ?
1621         symbols.getMonetaryDecimalSeparator() :
1622         symbols.getDecimalSeparator();

```

```

1623
1624     /* Per bug 4147706, DecimalFormat must respect the sign of numbers which
1625      * format as zero. This allows sensible computations and preserves
1626      * relations such as signum(1/x) = signum(x), where x is +Infinity or
1627      * -Infinity. Prior to this fix, we always formatted zero values as if
1628      * they were positive. Liu 7/6/98.
1629      */
1630     if (digitList.isZero()) {
1631         digitList.decimalAt = 0; // Normalize
1632     }
1633
1634     if (isNegative) {
1635         append(result, negativePrefix, delegate,
1636             getNegativePrefixFieldPositions(), Field.SIGN);
1637     } else {
1638         append(result, positivePrefix, delegate,
1639             getPositivePrefixFieldPositions(), Field.SIGN);
1640     }
1641
1642     if (useExponentialNotation) {
1643         int iFieldStart = result.length();
1644         int iFieldEnd = -1;
1645         int fFieldStart = -1;
1646
1647         // Minimum integer digits are handled in exponential format by
1648         // adjusting the exponent. For example, 0.01234 with 3 minimum
1649         // integer digits is "123.4E-4".
1650
1651         // Maximum integer digits are interpreted as indicating the
1652         // repeating range. This is useful for engineering notation, in
1653         // which the exponent is restricted to a multiple of 3. For
1654         // example, 0.01234 with 3 maximum integer digits is "12.34e-3".
1655         // If maximum integer digits are > 1 and are larger than
1656         // minimum integer digits, then minimum integer digits are
1657         // ignored.
1658         int exponent = digitList.decimalAt;
1659         int repeat = maxIntDigits;
1660         int minimumIntegerDigits = minIntDigits;
1661         if (repeat > 1 && repeat > minIntDigits) {
1662             // A repeating range is defined; adjust to it as follows.
1663             // If repeat == 3, we have 6,5,4=>3; 3,2,1=>0; 0,-1,-2=>-3;
1664             // -3,-4,-5=>-6, etc. This takes into account that the
1665             // exponent we have here is off by one from what we expect;
1666             // it is for the format 0.MMMMMx10^n.
1667             if (exponent >= 1) {
1668                 exponent = ((exponent - 1) / repeat) * repeat;
1669             } else {
1670                 // integer division rounds towards 0
1671                 exponent = ((exponent - repeat) / repeat) * repeat;
1672             }
1673             minimumIntegerDigits = 1;
1674         } else {
1675             // No repeating range is defined; use minimum integer digits.

```



```
1676         exponent -= minimumIntegerDigits;
1677     }
1678
1679     // We now output a minimum number of digits, and more if there
1680     // are more digits, up to the maximum number of digits. We
1681     // place the decimal point after the "integer" digits, which
1682     // are the first (decimalAt - exponent) digits.
1683     int minimumDigits = minIntDigits + minFraDigits;
1684     if (minimumDigits < 0) { // overflow?
1685         minimumDigits = Integer.MAX_VALUE;
1686     }
1687
1688     // The number of integer digits is handled specially if the number
1689     // is zero, since then there may be no digits.
1690     int integerDigits = digitList.isZero() ? minimumIntegerDigits :
1691         digitList.decimalAt - exponent;
1692     if (minimumDigits < integerDigits) {
1693         minimumDigits = integerDigits;
1694     }
1695     int totalDigits = digitList.count;
1696     if (minimumDigits > totalDigits) {
1697         totalDigits = minimumDigits;
1698     }
1699     boolean addedDecimalSeparator = false;
1700
1701     for (int i=0; i<totalDigits; ++i) {
1702         if (i == integerDigits) {
1703             // Record field information for caller.
1704             iFieldEnd = result.length();
1705
1706             result.append(decimal);
1707             addedDecimalSeparator = true;
1708
1709             // Record field information for caller.
1710             fFieldStart = result.length();
1711         }
1712         result.append((i < digitList.count) ?
1713             (char)(digitList.digits[i] + zeroDelta) :
1714             zero);
1715     }
1716
1717     if (decimalSeparatorAlwaysShown && totalDigits == integerDigits) {
1718         // Record field information for caller.
1719         iFieldEnd = result.length();
1720
1721         result.append(decimal);
1722         addedDecimalSeparator = true;
1723
1724         // Record field information for caller.
1725         fFieldStart = result.length();
1726     }
1727
1728     // Record field information
```

```

1729     if (iFieldEnd == -1) {
1730         iFieldEnd = result.length();
1731     }
1732     delegate.formatted(INTEGER_FIELD, Field.INTEGER, Field.INTEGER,
1733         iFieldStart, iFieldEnd, result);
1734     if (addedDecimalSeparator) {
1735         delegate.formatted(Field.DECIMAL_SEPARATOR,
1736             Field.DECIMAL_SEPARATOR,
1737             iFieldEnd, fFieldStart, result);
1738     }
1739     if (fFieldStart == -1) {
1740         fFieldStart = result.length();
1741     }
1742     delegate.formatted(FRACTION_FIELD, Field.FRACTION, Field.FRACTION,
1743         fFieldStart, result.length(), result);
1744
1745     // The exponent is output using the pattern-specified minimum
1746     // exponent digits. There is no maximum limit to the exponent
1747     // digits, since truncating the exponent would result in an
1748     // unacceptable inaccuracy.
1749     int fieldStart = result.length();
1750
1751     result.append(symbols.getExponentSeparator());
1752
1753     delegate.formatted(Field.EXPONENT_SYMBOL, Field.EXPONENT_SYMBOL,
1754         fieldStart, result.length(), result);
1755
1756     // For zero values, we force the exponent to zero. We
1757     // must do this here, and not earlier, because the value
1758     // is used to determine integer digit count above.
1759     if (digitList.isZero()) {
1760         exponent = 0;
1761     }
1762
1763     boolean negativeExponent = exponent < 0;
1764     if (negativeExponent) {
1765         exponent = -exponent;
1766         fieldStart = result.length();
1767         result.append(symbols.getMinusSign());
1768         delegate.formatted(Field.EXPONENT_SIGN, Field.EXPONENT_SIGN,
1769             fieldStart, result.length(), result);
1770     }
1771     digitList.set(negativeExponent, exponent);
1772
1773     int eFieldStart = result.length();
1774
1775     for (int i=digitList.decimalAt; i<minExponentDigits; ++i) {
1776         result.append(zero);
1777     }
1778     for (int i=0; i<digitList.decimalAt; ++i) {
1779         result.append((i < digitList.count) ?
1780             (char)(digitList.digits[i] + zeroDelta) : zero);
1781     }

```

```

1782         delegate.formatted(Field.EXPONENT, Field.EXPONENT, eFieldStart,
1783                             result.length(), result);
1784     } else {
1785         int iFieldStart = result.length();
1786
1787         // Output the integer portion. Here 'count' is the total
1788         // number of integer digits we will display, including both
1789         // leading zeros required to satisfy getMinimumIntegerDigits,
1790         // and actual digits present in the number.
1791         int count = minIntDigits;
1792         int digitIndex = 0; // Index into digitList.fDigits[]
1793         if (digitList.decimalAt > 0 && count < digitList.decimalAt) {
1794             count = digitList.decimalAt;
1795         }
1796
1797         // Handle the case where getMaximumIntegerDigits() is smaller
1798         // than the real number of integer digits. If this is so, we
1799         // output the least significant max integer digits. For example,
1800         // the value 1997 printed with 2 max integer digits is just "97".
1801         if (count > maxIntDigits) {
1802             count = maxIntDigits;
1803             digitIndex = digitList.decimalAt - count;
1804         }
1805
1806         int sizeBeforeIntegerPart = result.length();
1807         for (int i=count-1; i>=0; --i) {
1808             if (i < digitList.decimalAt && digitIndex < digitList.count) {
1809                 // Output a real digit
1810                 result.append((char)(digitList.digits[digitIndex++] + zeroDelta));
1811             } else {
1812                 // Output a leading zero
1813                 result.append(zero);
1814             }
1815
1816             // Output grouping separator if necessary. Don't output a
1817             // grouping separator if i==0 though; that's at the end of
1818             // the integer part.
1819             if (isGroupingUsed() && i>0 && (groupingSize != 0) &&
1820                 (i % groupingSize == 0)) {
1821                 int gStart = result.length();
1822                 result.append(grouping);
1823                 delegate.formatted(Field.GROUPING_SEPARATOR,
1824                                     Field.GROUPING_SEPARATOR, gStart,
1825                                     result.length(), result);
1826             }
1827         }
1828
1829         // Determine whether or not there are any printable fractional
1830         // digits. If we've used up the digits we know there aren't.
1831         boolean fractionPresent = (minFraDigits > 0) ||
1832             (!isInteger && digitIndex < digitList.count);
1833
1834         // If there is no fraction present, and we haven't printed any

```

```

1835     // integer digits, then print a zero.  Otherwise we won't print
1836     // _any_ digits, and we won't be able to parse this string.
1837     if (!fractionPresent && result.length() == sizeBeforeIntegerPart) {
1838         result.append(zero);
1839     }
1840
1841     delegate.formatted(INTEGER_FIELD, Field.INTEGER, Field.INTEGER,
1842                       iFieldStart, result.length(), result);
1843
1844     // Output the decimal separator if we always do so.
1845     int sStart = result.length();
1846     if (decimalSeparatorAlwaysShown || fractionPresent) {
1847         result.append(decimal);
1848     }
1849
1850     if (sStart != result.length()) {
1851         delegate.formatted(Field.DECIMAL_SEPARATOR,
1852                           Field.DECIMAL_SEPARATOR,
1853                           sStart, result.length(), result);
1854     }
1855     int fFieldStart = result.length();
1856
1857     for (int i=0; i < maxFraDigits; ++i) {
1858         // Here is where we escape from the loop.  We escape if we've
1859         // output the maximum fraction digits (specified in the for
1860         // expression above).
1861         // We also stop when we've output the minimum digits and either:
1862         // we have an integer, so there is no fractional stuff to
1863         // display, or we're out of significant digits.
1864         if (i >= minFraDigits &&
1865             (isInteger || digitIndex >= digitList.count)) {
1866             break;
1867         }
1868
1869         // Output leading fractional zeros.  These are zeros that come
1870         // after the decimal but before any significant digits.  These
1871         // are only output if abs(number being formatted) < 1.0.
1872         if (-1-i > (digitList.decimalAt-1)) {
1873             result.append(zero);
1874             continue;
1875         }
1876
1877         // Output a digit, if we have any precision left, or a
1878         // zero if we don't.  We don't want to output noise digits.
1879         if (!isInteger && digitIndex < digitList.count) {
1880             result.append((char)(digitList.digits[digitIndex++] + zeroDelta));
1881         } else {
1882             result.append(zero);
1883         }
1884     }
1885
1886     // Record field information for caller.
1887     delegate.formatted(FRACTION_FIELD, Field.FRACTION, Field.FRACTION,

```

```

1888         fFieldStart, result.length(), result);
1889     }
1890
1891     if (isNegative) {
1892         append(result, negativeSuffix, delegate,
1893             getNegativeSuffixFieldPositions(), Field.SIGN);
1894     } else {
1895         append(result, positiveSuffix, delegate,
1896             getPositiveSuffixFieldPositions(), Field.SIGN);
1897     }
1898
1899     return result;
1900 }
1901
1902 /**
1903  * Appends the String <code>string</code> to <code>result</code>.
1904  * <code>delegate</code> is notified of all the
1905  * <code>FieldPosition</code>s in <code>positions</code>.
1906  * <p>
1907  * If one of the <code>FieldPosition</code>s in <code>positions</code>
1908  * identifies a <code>SIGN</code> attribute, it is mapped to
1909  * <code>signAttribute</code>. This is used
1910  * to map the <code>SIGN</code> attribute to the <code>EXPONENT</code>
1911  * attribute as necessary.
1912  * <p>
1913  * This is used by <code>subformat</code> to add the prefix/suffix.
1914  */
1915 private void append(StringBuffer result, String string,
1916     FieldDelegate delegate,
1917     FieldPosition[] positions,
1918     Format.Field signAttribute) {
1919     int start = result.length();
1920
1921     if (string.length() > 0) {
1922         result.append(string);
1923         for (int counter = 0, max = positions.length; counter < max;
1924             counter++) {
1925             FieldPosition fp = positions[counter];
1926             Format.Field attribute = fp.getFieldAttribute();
1927
1928             if (attribute == Field.SIGN) {
1929                 attribute = signAttribute;
1930             }
1931             delegate.formatted(attribute, attribute,
1932                 start + fp.getBeginIndex(),
1933                 start + fp.getEndIndex(), result);
1934         }
1935     }
1936 }
1937
1938 /**
1939  * Parses text from a string to produce a <code>Number</code>.
1940  * <p>

```

```

1941 * The method attempts to parse text starting at the index given by
1942 * <code>pos</code>.
1943 * If parsing succeeds, then the index of <code>pos</code> is updated
1944 * to the index after the last character used (parsing does not necessarily
1945 * use all characters up to the end of the string), and the parsed
1946 * number is returned. The updated <code>pos</code> can be used to
1947 * indicate the starting point for the next call to this method.
1948 * If an error occurs, then the index of <code>pos</code> is not
1949 * changed, the error index of <code>pos</code> is set to the index of
1950 * the character where the error occurred, and null is returned.
1951 * <p>
1952 * The subclass returned depends on the value of {@link #isParseBigDecimal}
1953 * as well as on the string being parsed.
1954 * <ul>
1955 * <li>If <code>isParseBigDecimal()</code> is false (the default),
1956 *     most integer values are returned as <code>Long</code>
1957 *     objects, no matter how they are written: <code>"17"</code> and
1958 *     <code>"17.000"</code> both parse to <code>Long(17)</code>.
1959 *     Values that cannot fit into a <code>Long</code> are returned as
1960 *     <code>Double</code>s. This includes values with a fractional part,
1961 *     infinite values, <code>NaN</code>, and the value -0.0.
1962 *     <code>DecimalFormat</code> does not decide whether to
1963 *     return a <code>Double</code> or a <code>Long</code> based on the
1964 *     presence of a decimal separator in the source string. Doing so
1965 *     would prevent integers that overflow the mantissa of a double,
1966 *     such as <code>"-9,223,372,036,854,775,808.00"</code>, from being
1967 *     parsed accurately.
1968 *     <p>
1969 *     Callers may use the <code>Number</code> methods
1970 *     <code>doubleValue</code>, <code>longValue</code>, etc., to obtain
1971 *     the type they want.
1972 * <li>If <code>isParseBigDecimal()</code> is true, values are returned
1973 *     as <code>BigDecimal</code> objects. The values are the ones
1974 *     constructed by {@link java.math.BigDecimal#BigDecimal(String)}
1975 *     for corresponding strings in locale-independent format. The
1976 *     special cases negative and positive infinity and NaN are returned
1977 *     as <code>Double</code> instances holding the values of the
1978 *     corresponding <code>Double</code> constants.
1979 * </ul>
1980 * <p>
1981 * <code>DecimalFormat</code> parses all Unicode characters that represent
1982 * decimal digits, as defined by <code>Character.digit()</code>. In
1983 * addition, <code>DecimalFormat</code> also recognizes as digits the ten
1984 * consecutive characters starting with the localized zero digit defined in
1985 * the <code>DecimalFormatSymbols</code> object.
1986 *
1987 * @param text the string to be parsed
1988 * @param pos A <code>ParsePosition</code> object with index and error
1989 *            index information as described above.
1990 * @return the parsed value, or <code>null</code> if the parse fails
1991 * @exception NullPointerException if <code>text</code> or
1992 *            <code>pos</code> is null.
1993 */

```

```

1994     @Override
1995     public Number parse(String text, ParsePosition pos) {
1996         // special case NaN
1997         if (text.regionMatches(pos.index, symbols.getNaN(), 0, symbols.getNaN().length())) {
1998             pos.index = pos.index + symbols.getNaN().length();
1999             return new Double(Double.NaN);
2000         }
2001
2002         boolean[] status = new boolean[STATUS_LENGTH];
2003         if (!subparse(text, pos, positivePrefix, negativePrefix, digitList, false, status)) {
2004             return null;
2005         }
2006
2007         // special case INFINITY
2008         if (status[STATUS_INFINITE]) {
2009             if (status[STATUS_POSITIVE] == (multiplier >= 0)) {
2010                 return new Double(Double.POSITIVE_INFINITY);
2011             } else {
2012                 return new Double(Double.NEGATIVE_INFINITY);
2013             }
2014         }
2015
2016         if (multiplier == 0) {
2017             if (digitList.isZero()) {
2018                 return new Double(Double.NaN);
2019             } else if (status[STATUS_POSITIVE]) {
2020                 return new Double(Double.POSITIVE_INFINITY);
2021             } else {
2022                 return new Double(Double.NEGATIVE_INFINITY);
2023             }
2024         }
2025
2026         if (isParseBigDecimal()) {
2027             BigDecimal bigDecimalResult = digitList.getBigDecimal();
2028
2029             if (multiplier != 1) {
2030                 try {
2031                     bigDecimalResult = bigDecimalResult.divide(getBigDecimalMultiplier());
2032                 }
2033                 catch (ArithmeticException e) { // non-terminating decimal expansion
2034                     bigDecimalResult = bigDecimalResult.divide(getBigDecimalMultiplier(),
2035                         roundingMode);
2036                 }
2037
2038                 if (!status[STATUS_POSITIVE]) {
2039                     bigDecimalResult = bigDecimalResult.negate();
2040                 }
2041                 return bigDecimalResult;
2042             } else {
2043                 boolean gotDouble = true;
2044                 boolean gotLongMinimum = false;
2045                 double doubleResult = 0.0;

```

```

2046     long    longResult = 0;
2047
2048     // Finally, have DigitList parse the digits into a value.
2049     if (digitList.fitsIntoLong(status[STATUS_POSITIVE], isParseIntegerOnly())) {
2050         gotDouble = false;
2051         longResult = digitList.getLong();
2052         if (longResult < 0) { // got Long.MIN_VALUE
2053             gotLongMinimum = true;
2054         }
2055     } else {
2056         doubleResult = digitList.getDouble();
2057     }
2058
2059     // Divide by multiplier. We have to be careful here not to do
2060     // unneeded conversions between double and long.
2061     if (multiplier != 1) {
2062         if (gotDouble) {
2063             doubleResult /= multiplier;
2064         } else {
2065             // Avoid converting to double if we can
2066             if (longResult % multiplier == 0) {
2067                 longResult /= multiplier;
2068             } else {
2069                 doubleResult = ((double)longResult) / multiplier;
2070                 gotDouble = true;
2071             }
2072         }
2073     }
2074
2075     if (!status[STATUS_POSITIVE] && !gotLongMinimum) {
2076         doubleResult = -doubleResult;
2077         longResult = -longResult;
2078     }
2079
2080     // At this point, if we divided the result by the multiplier, the
2081     // result may fit into a long. We check for this case and return
2082     // a long if possible.
2083     // We must do this AFTER applying the negative (if appropriate)
2084     // in order to handle the case of LONG_MIN; otherwise, if we do
2085     // this with a positive value -LONG_MIN, the double is > 0, but
2086     // the long is < 0. We also must retain a double in the case of
2087     // -0.0, which will compare as == to a long 0 cast to a double
2088     // (bug 4162852).
2089     if (multiplier != 1 && gotDouble) {
2090         longResult = (long)doubleResult;
2091         gotDouble = ((doubleResult != (double)longResult) ||
2092                     (doubleResult == 0.0 && 1/doubleResult < 0.0)) &&
2093                     !isParseIntegerOnly();
2094     }
2095
2096     return gotDouble ?
2097         (Number)new Double(doubleResult) : (Number)new Long(longResult);
2098 }

```



```
2099     }
2100
2101     /**
2102      * Return a BigInteger multiplier.
2103      */
2104     private BigInteger getBigIntegerMultiplier() {
2105         if (bigIntegerMultiplier == null) {
2106             bigIntegerMultiplier = BigInteger.valueOf(multiplier);
2107         }
2108         return bigIntegerMultiplier;
2109     }
2110     private transient BigInteger bigIntegerMultiplier;
2111
2112     /**
2113      * Return a BigDecimal multiplier.
2114      */
2115     private BigDecimal getBigDecimalMultiplier() {
2116         if (bigDecimalMultiplier == null) {
2117             bigDecimalMultiplier = new BigDecimal(multiplier);
2118         }
2119         return bigDecimalMultiplier;
2120     }
2121     private transient BigDecimal bigDecimalMultiplier;
2122
2123     private static final int STATUS_INFINITE = 0;
2124     private static final int STATUS_POSITIVE = 1;
2125     private static final int STATUS_LENGTH = 2;
2126
2127     /**
2128      * Parse the given text into a number. The text is parsed beginning at
2129      * parsePosition, until an unparseable character is seen.
2130      * @param text The string to parse.
2131      * @param parsePosition The position at which to begin parsing. Upon
2132      * return, the first unparseable character.
2133      * @param digits The DigitList to set to the parsed value.
2134      * @param isExponent If true, parse an exponent. This means no
2135      * infinite values and integer only.
2136      * @param status Upon return contains boolean status flags indicating
2137      * whether the value was infinite and whether it was positive.
2138      */
2139     private final boolean subparse(String text, ParsePosition parsePosition,
2140                                   String positivePrefix, String negativePrefix,
2141                                   DigitList digits, boolean isExponent,
2142                                   boolean status[]) {
2143         int position = parsePosition.index;
2144         int oldStart = parsePosition.index;
2145         int backup;
2146         boolean gotPositive, gotNegative;
2147
2148         // check for positivePrefix; take longest
2149         gotPositive = text.regionMatches(position, positivePrefix, 0,
2150                                         positivePrefix.length());
2151         gotNegative = text.regionMatches(position, negativePrefix, 0,
```

```

2152         negativePrefix.length());
2153
2154     if (gotPositive && gotNegative) {
2155         if (positivePrefix.length() > negativePrefix.length()) {
2156             gotNegative = false;
2157         } else if (positivePrefix.length() < negativePrefix.length()) {
2158             gotPositive = false;
2159         }
2160     }
2161
2162     if (gotPositive) {
2163         position += positivePrefix.length();
2164     } else if (gotNegative) {
2165         position += negativePrefix.length();
2166     } else {
2167         parsePosition.errorIndex = position;
2168         return false;
2169     }
2170
2171     // process digits or Inf, find decimal position
2172     status[STATUS_INFINITE] = false;
2173     if (!isExponent && text.regionMatches(position, symbols.getInfinity(), 0,
2174         symbols.getInfinity().length())) {
2175         position += symbols.getInfinity().length();
2176         status[STATUS_INFINITE] = true;
2177     } else {
2178         // We now have a string of digits, possibly with grouping symbols,
2179         // and decimal points. We want to process these into a DigitList.
2180         // We don't want to put a bunch of leading zeros into the DigitList
2181         // though, so we keep track of the location of the decimal point,
2182         // put only significant digits into the DigitList, and adjust the
2183         // exponent as needed.
2184
2185         digits.decimalAt = digits.count = 0;
2186         char zero = symbols.getZeroDigit();
2187         char decimal = isCurrencyFormat ?
2188             symbols.getMonetaryDecimalSeparator() :
2189             symbols.getDecimalSeparator();
2190         char grouping = symbols.getGroupingSeparator();
2191         String exponentString = symbols.getExponentSeparator();
2192         boolean sawDecimal = false;
2193         boolean sawExponent = false;
2194         boolean sawDigit = false;
2195         int exponent = 0; // Set to the exponent value, if any
2196
2197         // We have to track digitCount ourselves, because digits.count will
2198         // pin when the maximum allowable digits is reached.
2199         int digitCount = 0;
2200
2201         backup = -1;
2202         for (; position < text.length(); ++position) {
2203             char ch = text.charAt(position);
2204

```

```

2205     /* We recognize all digit ranges, not only the Latin digit range
2206     * '0'..'9'. We do so by using the Character.digit() method,
2207     * which converts a valid Unicode digit to the range 0..9.
2208     *
2209     * The character 'ch' may be a digit. If so, place its value
2210     * from 0 to 9 in 'digit'. First try using the locale digit,
2211     * which may or MAY NOT be a standard Unicode digit range. If
2212     * this fails, try using the standard Unicode digit ranges by
2213     * calling Character.digit(). If this also fails, digit will
2214     * have a value outside the range 0..9.
2215     */
2216     int digit = ch - zero;
2217     if (digit < 0 || digit > 9) {
2218         digit = Character.digit(ch, 10);
2219     }
2220
2221     if (digit == 0) {
2222         // Cancel out backup setting (see grouping handler below)
2223         backup = -1; // Do this BEFORE continue statement below!!!
2224         sawDigit = true;
2225
2226         // Handle leading zeros
2227         if (digits.count == 0) {
2228             // Ignore leading zeros in integer part of number.
2229             if (!sawDecimal) {
2230                 continue;
2231             }
2232
2233             // If we have seen the decimal, but no significant
2234             // digits yet, then we account for leading zeros by
2235             // decrementing the digits.decimalAt into negative
2236             // values.
2237             --digits.decimalAt;
2238         } else {
2239             ++digitCount;
2240             digits.append((char)(digit + '0'));
2241         }
2242     } else if (digit > 0 && digit <= 9) { // [sic] digit==0 handled above
2243         sawDigit = true;
2244         ++digitCount;
2245         digits.append((char)(digit + '0'));
2246
2247         // Cancel out backup setting (see grouping handler below)
2248         backup = -1;
2249     } else if (!isExponent && ch == decimal) {
2250         // If we're only parsing integers, or if we ALREADY saw the
2251         // decimal, then don't parse this one.
2252         if (isParseIntegerOnly() || sawDecimal) {
2253             break;
2254         }
2255         digits.decimalAt = digitCount; // Not digits.count!
2256         sawDecimal = true;
2257     } else if (!isExponent && ch == grouping && isGroupingUsed()) {

```

```

2258         if (sawDecimal) {
2259             break;
2260         }
2261         // Ignore grouping characters, if we are using them, but
2262         // require that they be followed by a digit. Otherwise
2263         // we backup and reprocess them.
2264         backup = position;
2265     } else if (!isExponent && text.regionMatches(position, exponentString, 0,
        exponentString.length())
        && !sawExponent) {
2266         // Process the exponent by recursively calling this method.
2267         ParsePosition pos = new ParsePosition(position + exponentString.length());
2268         boolean[] stat = new boolean[STATUS_LENGTH];
2269         DigitList exponentDigits = new DigitList();
2270
2271         if (subparse(text, pos, "", Character.toString(symbols.getMinusSign()),
2272             exponentDigits, true, stat) &&
2273             exponentDigits.fitsIntoLong(stat[STATUS_POSITIVE], true)) {
2274             position = pos.index; // Advance past the exponent
2275             exponent = (int)exponentDigits.getLong();
2276             if (!stat[STATUS_POSITIVE]) {
2277                 exponent = -exponent;
2278             }
2279             sawExponent = true;
2280         }
2281         break; // Whether we fail or succeed, we exit this loop
2282     } else {
2283         break;
2284     }
2285 }
2286
2287 if (backup != -1) {
2288     position = backup;
2289 }
2290
2291 // If there was no decimal point we have an integer
2292 if (!sawDecimal) {
2293     digits.decimalAt = digitCount; // Not digits.count!
2294 }
2295
2296 // Adjust for exponent, if any
2297 digits.decimalAt += exponent;
2298
2299 // If none of the text string was recognized. For example, parse
2300 // "x" with pattern "#0.00" (return index and error index both 0)
2301 // parse "$" with pattern "$#0.00". (return index 0 and error
2302 // index 1).
2303 if (!sawDigit && digitCount == 0) {
2304     parsePosition.index = oldStart;
2305     parsePosition.errorIndex = oldStart;
2306     return false;
2307 }
2308 }

```

```

2309
2310     // check for suffix
2311     if (!isExponent) {
2312         if (gotPositive) {
2313             gotPositive = text.regionMatches(position, positiveSuffix, 0,
2314                                             positiveSuffix.length());
2315         }
2316         if (gotNegative) {
2317             gotNegative = text.regionMatches(position, negativeSuffix, 0,
2318                                             negativeSuffix.length());
2319         }
2320
2321         // if both match, take longest
2322         if (gotPositive && gotNegative) {
2323             if (positiveSuffix.length() > negativeSuffix.length()) {
2324                 gotNegative = false;
2325             } else if (positiveSuffix.length() < negativeSuffix.length()) {
2326                 gotPositive = false;
2327             }
2328         }
2329
2330         // fail if neither or both
2331         if (gotPositive == gotNegative) {
2332             parsePosition.errorIndex = position;
2333             return false;
2334         }
2335
2336         parsePosition.index = position +
2337             (gotPositive ? positiveSuffix.length() : negativeSuffix.length()); // mark success!
2338     } else {
2339         parsePosition.index = position;
2340     }
2341
2342     status[STATUS_POSITIVE] = gotPositive;
2343     if (parsePosition.index == oldStart) {
2344         parsePosition.errorIndex = position;
2345         return false;
2346     }
2347     return true;
2348 }
2349
2350 /**
2351  * Returns a copy of the decimal format symbols, which is generally not
2352  * changed by the programmer or user.
2353  * @return a copy of the desired DecimalFormatSymbols
2354  * @see java.text.DecimalFormatSymbols
2355  */
2356 public DecimalFormatSymbols getDecimalFormatSymbols() {
2357     try {
2358         // don't allow multiple references
2359         return (DecimalFormatSymbols) symbols.clone();
2360     } catch (Exception foo) {
2361         return null; // should never happen

```

```
2362     }
2363 }
2364
2365
2366 /**
2367  * Sets the decimal format symbols, which is generally not changed
2368  * by the programmer or user.
2369  * @param newSymbols desired DecimalFormatSymbols
2370  * @see java.text.DecimalFormatSymbols
2371  */
2372 public void setDecimalFormatSymbols(DecimalFormatSymbols newSymbols) {
2373     try {
2374         // don't allow multiple references
2375         symbols = (DecimalFormatSymbols) newSymbols.clone();
2376         expandAffixes();
2377         fastPathCheckNeeded = true;
2378     } catch (Exception foo) {
2379         // should never happen
2380     }
2381 }
2382
2383 /**
2384  * Get the positive prefix.
2385  * <P>Examples: +123, $123, sFr123
2386  *
2387  * @return the positive prefix
2388  */
2389 public String getPositivePrefix () {
2390     return positivePrefix;
2391 }
2392
2393 /**
2394  * Set the positive prefix.
2395  * <P>Examples: +123, $123, sFr123
2396  *
2397  * @param newValue the new positive prefix
2398  */
2399 public void setPositivePrefix (String newValue) {
2400     positivePrefix = newValue;
2401     posPrefixPattern = null;
2402     positivePrefixFieldPositions = null;
2403     fastPathCheckNeeded = true;
2404 }
2405
2406 /**
2407  * Returns the FieldPositions of the fields in the prefix used for
2408  * positive numbers. This is not used if the user has explicitly set
2409  * a positive prefix via <code>setPositivePrefix</code>. This is
2410  * lazily created.
2411  *
2412  * @return FieldPositions in positive prefix
2413  */
2414 private FieldPosition[] getPositivePrefixFieldPositions() {
```

```

2415     if (positivePrefixFieldPositions == null) {
2416         if (posPrefixPattern != null) {
2417             positivePrefixFieldPositions = expandAffix(posPrefixPattern);
2418         } else {
2419             positivePrefixFieldPositions = EmptyFieldPositionArray;
2420         }
2421     }
2422     return positivePrefixFieldPositions;
2423 }
2424
2425 /**
2426  * Get the negative prefix.
2427  * <P>Examples: -123, ($123) (with negative suffix), sFr-123
2428  *
2429  * @return the negative prefix
2430  */
2431 public String getNegativePrefix () {
2432     return negativePrefix;
2433 }
2434
2435 /**
2436  * Set the negative prefix.
2437  * <P>Examples: -123, ($123) (with negative suffix), sFr-123
2438  *
2439  * @param newValue the new negative prefix
2440  */
2441 public void setNegativePrefix (String newValue) {
2442     negativePrefix = newValue;
2443     negPrefixPattern = null;
2444     fastPathCheckNeeded = true;
2445 }
2446
2447 /**
2448  * Returns the FieldPositions of the fields in the prefix used for
2449  * negative numbers. This is not used if the user has explicitly set
2450  * a negative prefix via <code>setNegativePrefix</code>. This is
2451  * lazily created.
2452  *
2453  * @return FieldPositions in positive prefix
2454  */
2455 private FieldPosition[] getNegativePrefixFieldPositions() {
2456     if (negativePrefixFieldPositions == null) {
2457         if (negPrefixPattern != null) {
2458             negativePrefixFieldPositions = expandAffix(negPrefixPattern);
2459         } else {
2460             negativePrefixFieldPositions = EmptyFieldPositionArray;
2461         }
2462     }
2463     return negativePrefixFieldPositions;
2464 }
2465
2466 /**
2467  * Get the positive suffix.

```

```

2468     * <P>Example: 123%
2469     *
2470     * @return the positive suffix
2471     */
2472     public String getPositiveSuffix () {
2473         return positiveSuffix;
2474     }
2475
2476     /**
2477     * Set the positive suffix.
2478     * <P>Example: 123%
2479     *
2480     * @param newValue the new positive suffix
2481     */
2482     public void setPositiveSuffix (String newValue) {
2483         positiveSuffix = newValue;
2484         posSuffixPattern = null;
2485         fastPathCheckNeeded = true;
2486     }
2487
2488     /**
2489     * Returns the FieldPositions of the fields in the suffix used for
2490     * positive numbers. This is not used if the user has explicitly set
2491     * a positive suffix via <code>setPositiveSuffix</code>. This is
2492     * lazily created.
2493     *
2494     * @return FieldPositions in positive prefix
2495     */
2496     private FieldPosition[] getPositiveSuffixFieldPositions() {
2497         if (positiveSuffixFieldPositions == null) {
2498             if (posSuffixPattern != null) {
2499                 positiveSuffixFieldPositions = expandAffix(posSuffixPattern);
2500             } else {
2501                 positiveSuffixFieldPositions = EmptyFieldPositionArray;
2502             }
2503         }
2504         return positiveSuffixFieldPositions;
2505     }
2506
2507     /**
2508     * Get the negative suffix.
2509     * <P>Examples: -123%, ($123) (with positive suffixes)
2510     *
2511     * @return the negative suffix
2512     */
2513     public String getNegativeSuffix () {
2514         return negativeSuffix;
2515     }
2516
2517     /**
2518     * Set the negative suffix.
2519     * <P>Examples: 123%
2520     *

```



```

2521     * @param newValue the new negative suffix
2522     */
2523     public void setNegativeSuffix (String newValue) {
2524         negativeSuffix = newValue;
2525         negSuffixPattern = null;
2526         fastPathCheckNeeded = true;
2527     }
2528
2529     /**
2530     * Returns the FieldPositions of the fields in the suffix used for
2531     * negative numbers. This is not used if the user has explicitly set
2532     * a negative suffix via <code>setNegativeSuffix</code>. This is
2533     * lazily created.
2534     *
2535     * @return FieldPositions in positive prefix
2536     */
2537     private FieldPosition[] getNegativeSuffixFieldPositions() {
2538         if (negativeSuffixFieldPositions == null) {
2539             if (negSuffixPattern != null) {
2540                 negativeSuffixFieldPositions = expandAffix(negSuffixPattern);
2541             } else {
2542                 negativeSuffixFieldPositions = EmptyFieldPositionArray;
2543             }
2544         }
2545         return negativeSuffixFieldPositions;
2546     }
2547
2548     /**
2549     * Gets the multiplier for use in percent, per mille, and similar
2550     * formats.
2551     *
2552     * @return the multiplier
2553     * @see #setMultiplier(int)
2554     */
2555     public int getMultiplier () {
2556         return multiplier;
2557     }
2558
2559     /**
2560     * Sets the multiplier for use in percent, per mille, and similar
2561     * formats.
2562     * For a percent format, set the multiplier to 100 and the suffixes to
2563     * have '%' (for Arabic, use the Arabic percent sign).
2564     * For a per mille format, set the multiplier to 1000 and the suffixes to
2565     * have '&#92;u2030'.
2566     *
2567     * <P>Example: with multiplier 100, 1.23 is formatted as "123", and
2568     * "123" is parsed into 1.23.
2569     *
2570     * @param newValue the new multiplier
2571     * @see #getMultiplier
2572     */
2573     public void setMultiplier (int newValue) {

```

```

2574     multiplier = newValue;
2575     bigDecimalMultiplier = null;
2576     bigIntegerMultiplier = null;
2577     fastPathCheckNeeded = true;
2578 }
2579
2580 /**
2581  * {@inheritDoc}
2582  */
2583 @Override
2584 public void setGroupingUsed(boolean newValue) {
2585     super.setGroupingUsed(newValue);
2586     fastPathCheckNeeded = true;
2587 }
2588
2589 /**
2590  * Return the grouping size. Grouping size is the number of digits between
2591  * grouping separators in the integer portion of a number. For example,
2592  * in the number "123,456.78", the grouping size is 3.
2593  *
2594  * @return the grouping size
2595  * @see #setGroupingSize
2596  * @see java.text.NumberFormat#isGroupingUsed
2597  * @see java.text.DecimalFormatSymbols#getGroupingSeparator
2598  */
2599 public int getGroupingSize () {
2600     return groupingSize;
2601 }
2602
2603 /**
2604  * Set the grouping size. Grouping size is the number of digits between
2605  * grouping separators in the integer portion of a number. For example,
2606  * in the number "123,456.78", the grouping size is 3.
2607  * <br>
2608  * The value passed in is converted to a byte, which may lose information.
2609  *
2610  * @param newValue the new grouping size
2611  * @see #getGroupingSize
2612  * @see java.text.NumberFormat#setGroupingUsed
2613  * @see java.text.DecimalFormatSymbols#setGroupingSeparator
2614  */
2615 public void setGroupingSize (int newValue) {
2616     groupingSize = (byte)newValue;
2617     fastPathCheckNeeded = true;
2618 }
2619
2620 /**
2621  * Allows you to get the behavior of the decimal separator with integers.
2622  * (The decimal separator will always appear with decimals.)
2623  * <P>Example: Decimal ON: 12345 &rrarr; 12345.; OFF: 12345 &rrarr; 12345
2624  *
2625  * @return {@code true} if the decimal separator is always shown;
2626  *         {@code false} otherwise

```

```

2627     */
2628     public boolean isDecimalSeparatorAlwaysShown() {
2629         return decimalSeparatorAlwaysShown;
2630     }
2631
2632     /**
2633      * Allows you to set the behavior of the decimal separator with integers.
2634      * (The decimal separator will always appear with decimals.)
2635      * <P>Example: Decimal ON: 12345 &rrarr; 12345.; OFF: 12345 &rrarr; 12345
2636      *
2637      * @param newValue {@code true} if the decimal separator is always shown;
2638      *                  {@code false} otherwise
2639      */
2640     public void setDecimalSeparatorAlwaysShown(boolean newValue) {
2641         decimalSeparatorAlwaysShown = newValue;
2642         fastPathCheckNeeded = true;
2643     }
2644
2645     /**
2646      * Returns whether the {@link #parse(java.lang.String, java.text.ParsePosition)}
2647      * method returns <code>BigDecimal</code>. The default value is false.
2648      *
2649      * @return {@code true} if the parse method returns BigDecimal;
2650      *         {@code false} otherwise
2651      * @see #setParseBigDecimal
2652      * @since 1.5
2653      */
2654     public boolean isParseBigDecimal() {
2655         return parseBigDecimal;
2656     }
2657
2658     /**
2659      * Sets whether the {@link #parse(java.lang.String, java.text.ParsePosition)}
2660      * method returns <code>BigDecimal</code>.
2661      *
2662      * @param newValue {@code true} if the parse method returns BigDecimal;
2663      *                  {@code false} otherwise
2664      * @see #isParseBigDecimal
2665      * @since 1.5
2666      */
2667     public void setParseBigDecimal(boolean newValue) {
2668         parseBigDecimal = newValue;
2669     }
2670
2671     /**
2672      * Standard override; no change in semantics.
2673      */
2674     @Override
2675     public Object clone() {
2676         DecimalFormat other = (DecimalFormat) super.clone();
2677         other.symbols = (DecimalFormatSymbols) symbols.clone();
2678         other.digitList = (DigitList) digitList.clone();
2679     }

```

```

2680     // Fast-path is almost stateless algorithm. The only logical state is the
2681     // isFastPath flag. In addition fastPathCheckNeeded is a sentinel flag
2682     // that forces recalculation of all fast-path fields when set to true.
2683     //
2684     // There is thus no need to clone all the fast-path fields.
2685     // We just only need to set fastPathCheckNeeded to true when cloning,
2686     // and init fastPathData to null as if it were a truly new instance.
2687     // Every fast-path field will be recalculated (only once) at next usage of
2688     // fast-path algorithm.
2689     other.fastPathCheckNeeded = true;
2690     other.isFastPath = false;
2691     other.fastPathData = null;
2692
2693     return other;
2694 }
2695
2696 /**
2697  * Overrides equals
2698  */
2699 @Override
2700 public boolean equals(Object obj)
2701 {
2702     if (obj == null)
2703         return false;
2704     if (!super.equals(obj))
2705         return false; // super does class check
2706     DecimalFormat other = (DecimalFormat) obj;
2707     return ((posPrefixPattern == other.posPrefixPattern &&
2708             positivePrefix.equals(other.positivePrefix))
2709         || (posPrefixPattern != null &&
2710             posPrefixPattern.equals(other.posPrefixPattern)))
2711         && ((posSuffixPattern == other.posSuffixPattern &&
2712             positiveSuffix.equals(other.positiveSuffix))
2713         || (posSuffixPattern != null &&
2714             posSuffixPattern.equals(other.posSuffixPattern)))
2715         && ((negPrefixPattern == other.negPrefixPattern &&
2716             negativePrefix.equals(other.negativePrefix))
2717         || (negPrefixPattern != null &&
2718             negPrefixPattern.equals(other.negPrefixPattern)))
2719         && ((negSuffixPattern == other.negSuffixPattern &&
2720             negativeSuffix.equals(other.negativeSuffix))
2721         || (negSuffixPattern != null &&
2722             negSuffixPattern.equals(other.negSuffixPattern)))
2723         && multiplier == other.multiplier
2724         && groupingSize == other.groupingSize
2725         && decimalSeparatorAlwaysShown == other.decimalSeparatorAlwaysShown
2726         && parseBigDecimal == other.parseBigDecimal
2727         && useExponentialNotation == other.useExponentialNotation
2728         && (!useExponentialNotation ||
2729             minExponentDigits == other.minExponentDigits)
2730         && maximumIntegerDigits == other.maximumIntegerDigits
2731         && minimumIntegerDigits == other.minimumIntegerDigits
2732         && maximumFractionDigits == other.maximumFractionDigits

```

```

2733         && minimumFractionDigits == other.minimumFractionDigits
2734         && roundingMode == other.roundingMode
2735         && symbols.equals(other.symbols);
2736     }
2737
2738     /**
2739      * Overrides hashCode
2740      */
2741     @Override
2742     public int hashCode() {
2743         return super.hashCode() * 37 + positivePrefix.hashCode();
2744         // just enough fields for a reasonable distribution
2745     }
2746
2747     /**
2748      * Synthesizes a pattern string that represents the current state
2749      * of this Format object.
2750      *
2751      * @return a pattern string
2752      * @see #applyPattern
2753      */
2754     public String toPattern() {
2755         return toPattern( false );
2756     }
2757
2758     /**
2759      * Synthesizes a localized pattern string that represents the current
2760      * state of this Format object.
2761      *
2762      * @return a localized pattern string
2763      * @see #applyPattern
2764      */
2765     public String toLocalizedPattern() {
2766         return toPattern( true );
2767     }
2768
2769     /**
2770      * Expand the affix pattern strings into the expanded affix strings. If any
2771      * affix pattern string is null, do not expand it. This method should be
2772      * called any time the symbols or the affix patterns change in order to keep
2773      * the expanded affix strings up to date.
2774      */
2775     private void expandAffixes() {
2776         // Reuse one StringBuffer for better performance
2777         StringBuffer buffer = new StringBuffer();
2778         if (posPrefixPattern != null) {
2779             positivePrefix = expandAffix(posPrefixPattern, buffer);
2780             positivePrefixFieldPositions = null;
2781         }
2782         if (posSuffixPattern != null) {
2783             positiveSuffix = expandAffix(posSuffixPattern, buffer);
2784             positiveSuffixFieldPositions = null;
2785         }

```

```

2786     if (negPrefixPattern != null) {
2787         negativePrefix = expandAffix(negPrefixPattern, buffer);
2788         negativePrefixFieldPositions = null;
2789     }
2790     if (negSuffixPattern != null) {
2791         negativeSuffix = expandAffix(negSuffixPattern, buffer);
2792         negativeSuffixFieldPositions = null;
2793     }
2794 }
2795
2796 /**
2797  * Expand an affix pattern into an affix string. All characters in the
2798  * pattern are literal unless prefixed by QUOTE. The following characters
2799  * after QUOTE are recognized: PATTERN_PERCENT, PATTERN_PER_MILLE,
2800  * PATTERN_MINUS, and CURRENCY_SIGN. If CURRENCY_SIGN is doubled (QUOTE +
2801  * CURRENCY_SIGN + CURRENCY_SIGN), it is interpreted as an ISO 4217
2802  * currency code. Any other character after a QUOTE represents itself.
2803  * QUOTE must be followed by another character; QUOTE may not occur by
2804  * itself at the end of the pattern.
2805  *
2806  * @param pattern the non-null, possibly empty pattern
2807  * @param buffer a scratch StringBuffer; its contents will be lost
2808  * @return the expanded equivalent of pattern
2809  */
2810 private String expandAffix(String pattern, StringBuffer buffer) {
2811     buffer.setLength(0);
2812     for (int i=0; i<pattern.length(); ) {
2813         char c = pattern.charAt(i++);
2814         if (c == QUOTE) {
2815             c = pattern.charAt(i++);
2816             switch (c) {
2817                 case CURRENCY_SIGN:
2818                     if (i<pattern.length() &&
2819                         pattern.charAt(i) == CURRENCY_SIGN) {
2820                         ++i;
2821                         buffer.append(symbols.getInternationalCurrencySymbol());
2822                     } else {
2823                         buffer.append(symbols.getCurrencySymbol());
2824                     }
2825                     continue;
2826                 case PATTERN_PERCENT:
2827                     c = symbols.getPercent();
2828                     break;
2829                 case PATTERN_PER_MILLE:
2830                     c = symbols.getPerMill();
2831                     break;
2832                 case PATTERN_MINUS:
2833                     c = symbols.getMinusSign();
2834                     break;
2835             }
2836         }
2837         buffer.append(c);
2838     }

```

```

2839     return buffer.toString();
2840 }
2841
2842 /**
2843  * Expand an affix pattern into an array of FieldPositions describing
2844  * how the pattern would be expanded.
2845  * All characters in the
2846  * pattern are literal unless prefixed by QUOTE. The following characters
2847  * after QUOTE are recognized: PATTERN_PERCENT, PATTERN_PER_MILLE,
2848  * PATTERN_MINUS, and CURRENCY_SIGN. If CURRENCY_SIGN is doubled (QUOTE +
2849  * CURRENCY_SIGN + CURRENCY_SIGN), it is interpreted as an ISO 4217
2850  * currency code. Any other character after a QUOTE represents itself.
2851  * QUOTE must be followed by another character; QUOTE may not occur by
2852  * itself at the end of the pattern.
2853  *
2854  * @param pattern the non-null, possibly empty pattern
2855  * @return FieldPosition array of the resulting fields.
2856  */
2857 private FieldPosition[] expandAffix(String pattern) {
2858     ArrayList<FieldPosition> positions = null;
2859     int stringIndex = 0;
2860     for (int i=0; i<pattern.length(); ) {
2861         char c = pattern.charAt(i++);
2862         if (c == QUOTE) {
2863             int field = -1;
2864             Format.Field fieldID = null;
2865             c = pattern.charAt(i++);
2866             switch (c) {
2867                 case CURRENCY_SIGN:
2868                     String string;
2869                     if (i<pattern.length() &&
2870                         pattern.charAt(i) == CURRENCY_SIGN) {
2871                         ++i;
2872                         string = symbols.getInternationalCurrencySymbol();
2873                     } else {
2874                         string = symbols.getCurrencySymbol();
2875                     }
2876                     if (string.length() > 0) {
2877                         if (positions == null) {
2878                             positions = new ArrayList<>(2);
2879                         }
2880                         FieldPosition fp = new FieldPosition(Field.CURRENCY);
2881                         fp.setBeginIndex(stringIndex);
2882                         fp.setEndIndex(stringIndex + string.length());
2883                         positions.add(fp);
2884                         stringIndex += string.length();
2885                     }
2886                     continue;
2887                 case PATTERN_PERCENT:
2888                     c = symbols.getPercent();
2889                     field = -1;
2890                     fieldID = Field.PERCENT;
2891                     break;

```

```

2892         case PATTERN_PER_MILLE:
2893             c = symbols.getPerMill();
2894             field = -1;
2895             fieldID = Field.PERMILLE;
2896             break;
2897         case PATTERN_MINUS:
2898             c = symbols.getMinusSign();
2899             field = -1;
2900             fieldID = Field.SIGN;
2901             break;
2902     }
2903     if (fieldID != null) {
2904         if (positions == null) {
2905             positions = new ArrayList<>(2);
2906         }
2907         FieldPosition fp = new FieldPosition(fieldID, field);
2908         fp.setBeginIndex(stringIndex);
2909         fp.setEndIndex(stringIndex + 1);
2910         positions.add(fp);
2911     }
2912 }
2913 stringIndex++;
2914 }
2915 if (positions != null) {
2916     return positions.toArray(EmptyFieldPositionArray);
2917 }
2918 return EmptyFieldPositionArray;
2919 }
2920
2921 /**
2922  * Appends an affix pattern to the given StringBuffer, quoting special
2923  * characters as needed. Uses the internal affix pattern, if that exists,
2924  * or the literal affix, if the internal affix pattern is null. The
2925  * appended string will generate the same affix pattern (or literal affix)
2926  * when passed to toPattern().
2927  *
2928  * @param buffer the affix string is appended to this
2929  * @param affixPattern a pattern such as posPrefixPattern; may be null
2930  * @param expAffix a corresponding expanded affix, such as positivePrefix.
2931  * Ignored unless affixPattern is null. If affixPattern is null, then
2932  * expAffix is appended as a literal affix.
2933  * @param localized true if the appended pattern should contain localized
2934  * pattern characters; otherwise, non-localized pattern chars are appended
2935  */
2936 private void appendAffix(StringBuffer buffer, String affixPattern,
2937                         String expAffix, boolean localized) {
2938     if (affixPattern == null) {
2939         appendAffix(buffer, expAffix, localized);
2940     } else {
2941         int i;
2942         for (int pos=0; pos<affixPattern.length(); pos=i) {
2943             i = affixPattern.indexOf(QUOTE, pos);
2944             if (i < 0) {

```



```

2945         appendAffix(buffer, affixPattern.substring(pos), localized);
2946         break;
2947     }
2948     if (i > pos) {
2949         appendAffix(buffer, affixPattern.substring(pos, i), localized);
2950     }
2951     char c = affixPattern.charAt(++i);
2952     ++i;
2953     if (c == QUOTE) {
2954         buffer.append(c);
2955         // Fall through and append another QUOTE below
2956     } else if (c == CURRENCY_SIGN &&
2957         i < affixPattern.length() &&
2958         affixPattern.charAt(i) == CURRENCY_SIGN) {
2959         ++i;
2960         buffer.append(c);
2961         // Fall through and append another CURRENCY_SIGN below
2962     } else if (localized) {
2963         switch (c) {
2964             case PATTERN_PERCENT:
2965                 c = symbols.getPercent();
2966                 break;
2967             case PATTERN_PER_MILLE:
2968                 c = symbols.getPerMill();
2969                 break;
2970             case PATTERN_MINUS:
2971                 c = symbols.getMinusSign();
2972                 break;
2973         }
2974     }
2975     buffer.append(c);
2976 }
2977 }
2978 }
2979
2980 /**
2981  * Append an affix to the given StringBuffer, using quotes if
2982  * there are special characters. Single quotes themselves must be
2983  * escaped in either case.
2984  */
2985 private void appendAffix(StringBuffer buffer, String affix, boolean localized) {
2986     boolean needQuote;
2987     if (localized) {
2988         needQuote = affix.indexOf(symbols.getZeroDigit()) >= 0
2989             || affix.indexOf(symbols.getGroupingSeparator()) >= 0
2990             || affix.indexOf(symbols.getDecimalSeparator()) >= 0
2991             || affix.indexOf(symbols.getPercent()) >= 0
2992             || affix.indexOf(symbols.getPerMill()) >= 0
2993             || affix.indexOf(symbols.getDigit()) >= 0
2994             || affix.indexOf(symbols.getPatternSeparator()) >= 0
2995             || affix.indexOf(symbols.getMinusSign()) >= 0
2996             || affix.indexOf(CURRENCY_SIGN) >= 0;
2997     } else {

```

```

2998         needQuote = affix.indexOf(PATTERN_ZERO_DIGIT) >= 0
2999             || affix.indexOf(PATTERN_GROUPING_SEPARATOR) >= 0
3000             || affix.indexOf(PATTERN_DECIMAL_SEPARATOR) >= 0
3001             || affix.indexOf(PATTERN_PERCENT) >= 0
3002             || affix.indexOf(PATTERN_PER_MILLE) >= 0
3003             || affix.indexOf(PATTERN_DIGIT) >= 0
3004             || affix.indexOf(PATTERN_SEPARATOR) >= 0
3005             || affix.indexOf(PATTERN_MINUS) >= 0
3006             || affix.indexOf(CURRENCY_SIGN) >= 0;
3007     }
3008     if (needQuote) buffer.append('\\');
3009     if (affix.indexOf('\\') < 0) buffer.append(affix);
3010     else {
3011         for (int j=0; j<affix.length(); ++j) {
3012             char c = affix.charAt(j);
3013             buffer.append(c);
3014             if (c == '\\') buffer.append(c);
3015         }
3016     }
3017     if (needQuote) buffer.append('\\');
3018 }
3019
3020 /**
3021  * Does the real work of generating a pattern. */
3022 private String toPattern(boolean localized) {
3023     StringBuffer result = new StringBuffer();
3024     for (int j = 1; j >= 0; --j) {
3025         if (j == 1)
3026             appendAffix(result, posPrefixPattern, positivePrefix, localized);
3027         else appendAffix(result, negPrefixPattern, negativePrefix, localized);
3028         int i;
3029         int digitCount = useExponentialNotation
3030             ? getMaximumIntegerDigits()
3031             : Math.max(groupingSize, getMinimumIntegerDigits()+1);
3032         for (i = digitCount; i > 0; --i) {
3033             if (i != digitCount && isGroupingUsed() && groupingSize != 0 &&
3034                 i % groupingSize == 0) {
3035                 result.append(localized ? symbols.getGroupingSeparator() :
3036                     PATTERN_GROUPING_SEPARATOR);
3037             }
3038             result.append(i <= getMinimumIntegerDigits()
3039                 ? (localized ? symbols.getZeroDigit() : PATTERN_ZERO_DIGIT)
3040                 : (localized ? symbols.getDigit() : PATTERN_DIGIT));
3041         }
3042         if (getMaximumFractionDigits() > 0 || decimalSeparatorAlwaysShown)
3043             result.append(localized ? symbols.getDecimalSeparator() :
3044                 PATTERN_DECIMAL_SEPARATOR);
3045         for (i = 0; i < getMaximumFractionDigits(); ++i) {
3046             if (i < getMinimumFractionDigits()) {
3047                 result.append(localized ? symbols.getZeroDigit() :
3048                     PATTERN_ZERO_DIGIT);
3049             } else {
3050                 result.append(localized ? symbols.getDigit() :

```

```

3051         PATTERN_DIGIT);
3052     }
3053 }
3054 if (useExponentialNotation)
3055 {
3056     result.append(localized ? symbols.getExponentSeparator() :
3057         PATTERN_EXPONENT);
3058     for (i=0; i<minExponentDigits; ++i)
3059         result.append(localized ? symbols.getZeroDigit() :
3060             PATTERN_ZERO_DIGIT);
3061 }
3062 if (j == 1) {
3063     appendAffix(result, posSuffixPattern, positiveSuffix, localized);
3064     if ((negSuffixPattern == posSuffixPattern && // n == p == null
3065         negativeSuffix.equals(positiveSuffix))
3066         || (negSuffixPattern != null &&
3067             negSuffixPattern.equals(posSuffixPattern))) {
3068         if ((negPrefixPattern != null && posPrefixPattern != null &&
3069             negPrefixPattern.equals("'" + posPrefixPattern)) ||
3070             (negPrefixPattern == posPrefixPattern && // n == p == null
3071                 negativePrefix.equals(symbols.getMinusSign() + positivePrefix)))
3072             break;
3073     }
3074     result.append(localized ? symbols.getPatternSeparator() :
3075         PATTERN_SEPARATOR);
3076 } else appendAffix(result, negSuffixPattern, negativeSuffix, localized);
3077 }
3078 return result.toString();
3079 }
3080
3081 /**
3082  * Apply the given pattern to this Format object. A pattern is a
3083  * short-hand specification for the various formatting properties.
3084  * These properties can also be changed individually through the
3085  * various setter methods.
3086  * <p>
3087  * There is no limit to integer digits set
3088  * by this routine, since that is the typical end-user desire;
3089  * use setMaximumInteger if you want to set a real value.
3090  * For negative numbers, use a second pattern, separated by a semicolon
3091  * <P>Example <code>"#, #00.0#"</code> &rarr; 1,234.56
3092  * <P>This means a minimum of 2 integer digits, 1 fraction digit, and
3093  * a maximum of 2 fraction digits.
3094  * <p>Example: <code>"#, #00.0#; (#, #00.0#)"</code> for negatives in
3095  * parentheses.
3096  * <p>In negative patterns, the minimum and maximum counts are ignored;
3097  * these are presumed to be set in the positive pattern.
3098  *
3099  * @param pattern a new pattern
3100  * @exception NullPointerException if <code>pattern</code> is null
3101  * @exception IllegalArgumentException if the given pattern is invalid.
3102  */
3103 public void applyPattern(String pattern) {

```

```

3104     applyPattern(pattern, false);
3105 }
3106
3107 /**
3108  * Apply the given pattern to this Format object. The pattern
3109  * is assumed to be in a localized notation. A pattern is a
3110  * short-hand specification for the various formatting properties.
3111  * These properties can also be changed individually through the
3112  * various setter methods.
3113  * <p>
3114  * There is no limit to integer digits set
3115  * by this routine, since that is the typical end-user desire;
3116  * use setMaximumInteger if you want to set a real value.
3117  * For negative numbers, use a second pattern, separated by a semicolon
3118  * <P>Example <code>"#, #00.0#"</code> &rarr; 1,234.56
3119  * <P>This means a minimum of 2 integer digits, 1 fraction digit, and
3120  * a maximum of 2 fraction digits.
3121  * <p>Example: <code>"#, #00.0#; (#, #00.0#)"</code> for negatives in
3122  * parentheses.
3123  * <p>In negative patterns, the minimum and maximum counts are ignored;
3124  * these are presumed to be set in the positive pattern.
3125  *
3126  * @param pattern a new pattern
3127  * @exception NullPointerException if <code>pattern</code> is null
3128  * @exception IllegalArgumentException if the given pattern is invalid.
3129  */
3130 public void applyLocalizedPattern(String pattern) {
3131     applyPattern(pattern, true);
3132 }
3133
3134 /**
3135  * Does the real work of applying a pattern.
3136  */
3137 private void applyPattern(String pattern, boolean localized) {
3138     char zeroDigit      = PATTERN_ZERO_DIGIT;
3139     char groupingSeparator = PATTERN_GROUPING_SEPARATOR;
3140     char decimalSeparator = PATTERN_DECIMAL_SEPARATOR;
3141     char percent         = PATTERN_PERCENT;
3142     char perMill         = PATTERN_PER_MILLE;
3143     char digit           = PATTERN_DIGIT;
3144     char separator       = PATTERN_SEPARATOR;
3145     String exponent      = PATTERN_EXPONENT;
3146     char minus           = PATTERN_MINUS;
3147     if (localized) {
3148         zeroDigit      = symbols.getZeroDigit();
3149         groupingSeparator = symbols.getGroupingSeparator();
3150         decimalSeparator = symbols.getDecimalSeparator();
3151         percent         = symbols.getPercent();
3152         perMill         = symbols.getPerMill();
3153         digit           = symbols.getDigit();
3154         separator       = symbols.getPatternSeparator();
3155         exponent        = symbols.getExponentSeparator();
3156         minus           = symbols.getMinusSign();

```

```

3157     }
3158     boolean gotNegative = false;
3159     decimalSeparatorAlwaysShown = false;
3160     isCurrencyFormat = false;
3161     useExponentialNotation = false;
3162
3163     // Two variables are used to record the subrange of the pattern
3164     // occupied by phase 1. This is used during the processing of the
3165     // second pattern (the one representing negative numbers) to ensure
3166     // that no deviation exists in phase 1 between the two patterns.
3167     int phaseOneStart = 0;
3168     int phaseOneLength = 0;
3169
3170     int start = 0;
3171     for (int j = 1; j >= 0 && start < pattern.length(); --j) {
3172         boolean inQuote = false;
3173         StringBuffer prefix = new StringBuffer();
3174         StringBuffer suffix = new StringBuffer();
3175         int decimalPos = -1;
3176         int multiplier = 1;
3177         int digitLeftCount = 0, zeroDigitCount = 0, digitRightCount = 0;
3178         byte groupingCount = -1;
3179
3180         // The phase ranges from 0 to 2. Phase 0 is the prefix. Phase 1 is
3181         // the section of the pattern with digits, decimal separator,
3182         // grouping characters. Phase 2 is the suffix. In phases 0 and 2,
3183         // percent, per mille, and currency symbols are recognized and
3184         // translated. The separation of the characters into phases is
3185         // strictly enforced; if phase 1 characters are to appear in the
3186         // suffix, for example, they must be quoted.
3187         int phase = 0;
3188
3189         // The affix is either the prefix or the suffix.
3190         StringBuffer affix = prefix;
3191
3192         for (int pos = start; pos < pattern.length(); ++pos) {
3193             char ch = pattern.charAt(pos);
3194             switch (phase) {
3195                 case 0:
3196                 case 2:
3197                     // Process the prefix / suffix characters
3198                     if (inQuote) {
3199                         // A quote within quotes indicates either the closing
3200                         // quote or two quotes, which is a quote literal. That
3201                         // is, we have the second quote in 'do' or 'don't'.
3202                         if (ch == QUOTE) {
3203                             if ((pos+1) < pattern.length() &&
3204                                 pattern.charAt(pos+1) == QUOTE) {
3205                                 ++pos;
3206                                 affix.append("''"); // 'don't'
3207                             } else {
3208                                 inQuote = false; // 'do'
3209                             }

```

```

3210         continue;
3211     }
3212 } else {
3213     // Process unquoted characters seen in prefix or suffix
3214     // phase.
3215     if (ch == digit ||
3216         ch == zeroDigit ||
3217         ch == groupingSeparator ||
3218         ch == decimalSeparator) {
3219         phase = 1;
3220         if (j == 1) {
3221             phaseOneStart = pos;
3222         }
3223         --pos; // Reprocess this character
3224         continue;
3225     } else if (ch == CURRENCY_SIGN) {
3226         // Use lookahead to determine if the currency sign
3227         // is doubled or not.
3228         boolean doubled = (pos + 1) < pattern.length() &&
3229             pattern.charAt(pos + 1) == CURRENCY_SIGN;
3230         if (doubled) { // Skip over the doubled character
3231             ++pos;
3232         }
3233         isCurrencyFormat = true;
3234         affix.append(doubled ? "'\u00A4\u00A4" : "'\u00A4");
3235         continue;
3236     } else if (ch == QUOTE) {
3237         // A quote outside quotes indicates either the
3238         // opening quote or two quotes, which is a quote
3239         // literal. That is, we have the first quote in 'do'
3240         // or o'clock.
3241         if (ch == QUOTE) {
3242             if ((pos+1) < pattern.length() &&
3243                 pattern.charAt(pos+1) == QUOTE) {
3244                 ++pos;
3245                 affix.append("''"); // o'clock
3246             } else {
3247                 inQuote = true; // 'do'
3248             }
3249             continue;
3250         }
3251     } else if (ch == separator) {
3252         // Don't allow separators before we see digit
3253         // characters of phase 1, and don't allow separators
3254         // in the second pattern (j == 0).
3255         if (phase == 0 || j == 0) {
3256             throw new IllegalArgumentException("Unquoted special character '" +
3257                 ch + "' in pattern \"" + pattern + "\"");
3258         }
3259         start = pos + 1;
3260         pos = pattern.length();
3261         continue;
3262     }

```

```

3263
3264         // Next handle characters which are appended directly.
3265         else if (ch == percent) {
3266             if (multiplier != 1) {
3267                 throw new IllegalArgumentException("Too many percent/per mille
3268                     characters in pattern \"" +
3269                     pattern + "'");
3270             }
3271             multiplier = 100;
3272             affix.append("%");
3273             continue;
3274         } else if (ch == perMill) {
3275             if (multiplier != 1) {
3276                 throw new IllegalArgumentException("Too many percent/per mille
3277                     characters in pattern \"" +
3278                     pattern + "'");
3279             }
3280             multiplier = 1000;
3281             affix.append("\u2030");
3282             continue;
3283         } else if (ch == minus) {
3284             affix.append("-");
3285             continue;
3286         }
3287         // Note that if we are within quotes, or if this is an
3288         // unquoted, non-special character, then we usually fall
3289         // through to here.
3290         affix.append(ch);
3291         break;
3292
3293     case 1:
3294         // Phase one must be identical in the two sub-patterns. We
3295         // enforce this by doing a direct comparison. While
3296         // processing the first sub-pattern, we just record its
3297         // length. While processing the second, we compare
3298         // characters.
3299         if (j == 1) {
3300             ++phaseOneLength;
3301         } else {
3302             if (--phaseOneLength == 0) {
3303                 phase = 2;
3304                 affix = suffix;
3305             }
3306             continue;
3307         }
3308
3309         // Process the digits, decimal, and grouping characters. We
3310         // record five pieces of information. We expect the digits
3311         // to occur in the pattern #####0000.####, and we record the
3312         // number of left digits, zero (central) digits, and right
3313         // digits. The position of the last grouping character is
3314         // recorded (should be somewhere within the first two blocks

```

```

3314         // of characters), as is the position of the decimal point,
3315         // if any (should be in the zero digits). If there is no
3316         // decimal point, then there should be no right digits.
3317         if (ch == digit) {
3318             if (zeroDigitCount > 0) {
3319                 ++digitRightCount;
3320             } else {
3321                 ++digitLeftCount;
3322             }
3323             if (groupingCount >= 0 && decimalPos < 0) {
3324                 ++groupingCount;
3325             }
3326         } else if (ch == zeroDigit) {
3327             if (digitRightCount > 0) {
3328                 throw new IllegalArgumentException("Unexpected '0' in pattern \"" +
3329                     pattern + "'");
3330             }
3331             ++zeroDigitCount;
3332             if (groupingCount >= 0 && decimalPos < 0) {
3333                 ++groupingCount;
3334             }
3335         } else if (ch == groupingSeparator) {
3336             groupingCount = 0;
3337         } else if (ch == decimalSeparator) {
3338             if (decimalPos >= 0) {
3339                 throw new IllegalArgumentException("Multiple decimal separators in
3340                     pattern \"" +
3341                     pattern + "'");
3342             }
3343             decimalPos = digitLeftCount + zeroDigitCount + digitRightCount;
3344         } else if (pattern.regionMatches(pos, exponent, 0, exponent.length())) {
3345             if (useExponentialNotation) {
3346                 throw new IllegalArgumentException("Multiple exponential " +
3347                     "symbols in pattern \"" + pattern + "'");
3348             }
3349             useExponentialNotation = true;
3350             minExponentDigits = 0;
3351
3352             // Use lookahead to parse out the exponential part
3353             // of the pattern, then jump into phase 2.
3354             pos = pos + exponent.length();
3355             while (pos < pattern.length() &&
3356                 pattern.charAt(pos) == zeroDigit) {
3357                 ++minExponentDigits;
3358                 ++phaseOneLength;
3359                 ++pos;
3360             }
3361
3362             if ((digitLeftCount + zeroDigitCount) < 1 ||
3363                 minExponentDigits < 1) {
3364                 throw new IllegalArgumentException("Malformed exponential " +
3365                     "pattern \"" + pattern + "'");

```



```

3366
3367         // Transition to phase 2
3368         phase = 2;
3369         affix = suffix;
3370         --pos;
3371         continue;
3372     } else {
3373         phase = 2;
3374         affix = suffix;
3375         --pos;
3376         --phaseOneLength;
3377         continue;
3378     }
3379     break;
3380 }
3381 }
3382
3383 // Handle patterns with no '0' pattern character. These patterns
3384 // are legal, but must be interpreted. "##.###" -> "#0.###".
3385 // ".###" -> ".0##".
3386 /* We allow patterns of the form "####" to produce a zeroDigitCount
3387 * of zero (got that?); although this seems like it might make it
3388 * possible for format() to produce empty strings, format() checks
3389 * for this condition and outputs a zero digit in this situation.
3390 * Having a zeroDigitCount of zero yields a minimum integer digits
3391 * of zero, which allows proper round-trip patterns. That is, we
3392 * don't want "#" to become "#0" when toPattern() is called (even
3393 * though that's what it really is, semantically).
3394 */
3395 if (zeroDigitCount == 0 && digitLeftCount > 0 && decimalPos >= 0) {
3396     // Handle "##.###" and "##." and ".###"
3397     int n = decimalPos;
3398     if (n == 0) { // Handle ".###"
3399         ++n;
3400     }
3401     digitRightCount = digitLeftCount - n;
3402     digitLeftCount = n - 1;
3403     zeroDigitCount = 1;
3404 }
3405
3406 // Do syntax checking on the digits.
3407 if ((decimalPos < 0 && digitRightCount > 0) ||
3408     (decimalPos >= 0 && (decimalPos < digitLeftCount ||
3409         decimalPos > (digitLeftCount + zeroDigitCount))) ||
3410     groupingCount == 0 || inQuote) {
3411     throw new IllegalArgumentException("Malformed pattern \"" +
3412         pattern + "\"");
3413 }
3414
3415 if (j == 1) {
3416     posPrefixPattern = prefix.toString();
3417     posSuffixPattern = suffix.toString();
3418     negPrefixPattern = posPrefixPattern; // assume these for now

```

```

3419         negSuffixPattern = posSuffixPattern;
3420         int digitTotalCount = digitLeftCount + zeroDigitCount + digitRightCount;
3421         /* The effectiveDecimalPos is the position the decimal is at or
3422          * would be at if there is no decimal. Note that if decimalPos<0,
3423          * then digitTotalCount == digitLeftCount + zeroDigitCount.
3424          */
3425         int effectiveDecimalPos = decimalPos >= 0 ?
3426             decimalPos : digitTotalCount;
3427         setMinimumIntegerDigits(effectiveDecimalPos - digitLeftCount);
3428         setMaximumIntegerDigits(useExponentialNotation ?
3429             digitLeftCount + getMinimumIntegerDigits() :
3430             MAXIMUM_INTEGER_DIGITS);
3431         setMaximumFractionDigits(decimalPos >= 0 ?
3432             (digitTotalCount - decimalPos) : 0);
3433         setMinimumFractionDigits(decimalPos >= 0 ?
3434             (digitLeftCount + zeroDigitCount - decimalPos) : 0);
3435         setGroupingUsed(groupingCount > 0);
3436         this.groupingSize = (groupingCount > 0) ? groupingCount : 0;
3437         this.multiplier = multiplier;
3438         setDecimalSeparatorAlwaysShown(decimalPos == 0 ||
3439             decimalPos == digitTotalCount);
3440     } else {
3441         negPrefixPattern = prefix.toString();
3442         negSuffixPattern = suffix.toString();
3443         gotNegative = true;
3444     }
3445 }
3446
3447 if (pattern.length() == 0) {
3448     posPrefixPattern = posSuffixPattern = "";
3449     setMinimumIntegerDigits(0);
3450     setMaximumIntegerDigits(MAXIMUM_INTEGER_DIGITS);
3451     setMinimumFractionDigits(0);
3452     setMaximumFractionDigits(MAXIMUM_FRACTION_DIGITS);
3453 }
3454
3455 // If there was no negative pattern, or if the negative pattern is
3456 // identical to the positive pattern, then prepend the minus sign to
3457 // the positive pattern to form the negative pattern.
3458 if (!gotNegative ||
3459     (negPrefixPattern.equals(posPrefixPattern)
3460      && negSuffixPattern.equals(posSuffixPattern))) {
3461     negSuffixPattern = posSuffixPattern;
3462     negPrefixPattern = "'-' + posPrefixPattern;
3463 }
3464
3465 expandAffixes();
3466 }
3467
3468 /**
3469  * Sets the maximum number of digits allowed in the integer portion of a
3470  * number.
3471  * For formatting numbers other than <code>BigInteger</code> and

```

```

3472 * <code>BigDecimal</code> objects, the lower of <code>newValue</code> and
3473 * 309 is used. Negative input values are replaced with 0.
3474 * @see NumberFormat#setMaximumIntegerDigits
3475 */
3476 @Override
3477 public void setMaximumIntegerDigits(int newValue) {
3478     maximumIntegerDigits = Math.min(Math.max(0, newValue), MAXIMUM_INTEGER_DIGITS);
3479     super.setMaximumIntegerDigits((maximumIntegerDigits > DOUBLE_INTEGER_DIGITS) ?
3480         DOUBLE_INTEGER_DIGITS : maximumIntegerDigits);
3481     if (minimumIntegerDigits > maximumIntegerDigits) {
3482         minimumIntegerDigits = maximumIntegerDigits;
3483         super.setMinimumIntegerDigits((minimumIntegerDigits > DOUBLE_INTEGER_DIGITS) ?
3484             DOUBLE_INTEGER_DIGITS : minimumIntegerDigits);
3485     }
3486     fastPathCheckNeeded = true;
3487 }
3488
3489 /**
3490  * Sets the minimum number of digits allowed in the integer portion of a
3491  * number.
3492  * For formatting numbers other than <code>BigInteger</code> and
3493  * <code>BigDecimal</code> objects, the lower of <code>newValue</code> and
3494  * 309 is used. Negative input values are replaced with 0.
3495  * @see NumberFormat#setMinimumIntegerDigits
3496  */
3497 @Override
3498 public void setMinimumIntegerDigits(int newValue) {
3499     minimumIntegerDigits = Math.min(Math.max(0, newValue), MAXIMUM_INTEGER_DIGITS);
3500     super.setMinimumIntegerDigits((minimumIntegerDigits > DOUBLE_INTEGER_DIGITS) ?
3501         DOUBLE_INTEGER_DIGITS : minimumIntegerDigits);
3502     if (minimumIntegerDigits > maximumIntegerDigits) {
3503         maximumIntegerDigits = minimumIntegerDigits;
3504         super.setMaximumIntegerDigits((maximumIntegerDigits > DOUBLE_INTEGER_DIGITS) ?
3505             DOUBLE_INTEGER_DIGITS : maximumIntegerDigits);
3506     }
3507     fastPathCheckNeeded = true;
3508 }
3509
3510 /**
3511  * Sets the maximum number of digits allowed in the fraction portion of a
3512  * number.
3513  * For formatting numbers other than <code>BigInteger</code> and
3514  * <code>BigDecimal</code> objects, the lower of <code>newValue</code> and
3515  * 340 is used. Negative input values are replaced with 0.
3516  * @see NumberFormat#setMaximumFractionDigits
3517  */
3518 @Override
3519 public void setMaximumFractionDigits(int newValue) {
3520     maximumFractionDigits = Math.min(Math.max(0, newValue), MAXIMUM_FRACTION_DIGITS);
3521     super.setMaximumFractionDigits((maximumFractionDigits > DOUBLE_FRACTION_DIGITS) ?
3522         DOUBLE_FRACTION_DIGITS : maximumFractionDigits);
3523     if (minimumFractionDigits > maximumFractionDigits) {
3524         minimumFractionDigits = maximumFractionDigits;

```

```

3525         super.setMinimumFractionDigits((minimumFractionDigits > DOUBLE_FRACTION_DIGITS) ?
3526             DOUBLE_FRACTION_DIGITS : minimumFractionDigits);
3527     }
3528     fastPathCheckNeeded = true;
3529 }
3530
3531 /**
3532  * Sets the minimum number of digits allowed in the fraction portion of a
3533  * number.
3534  * For formatting numbers other than <code>BigInteger</code> and
3535  * <code>BigDecimal</code> objects, the lower of <code>newValue</code> and
3536  * 340 is used. Negative input values are replaced with 0.
3537  * @see NumberFormat#setMinimumFractionDigits
3538  */
3539 @Override
3540 public void setMinimumFractionDigits(int newValue) {
3541     minimumFractionDigits = Math.min(Math.max(0, newValue), MAXIMUM_FRACTION_DIGITS);
3542     super.setMinimumFractionDigits((minimumFractionDigits > DOUBLE_FRACTION_DIGITS) ?
3543         DOUBLE_FRACTION_DIGITS : minimumFractionDigits);
3544     if (minimumFractionDigits > maximumFractionDigits) {
3545         maximumFractionDigits = minimumFractionDigits;
3546         super.setMaximumFractionDigits((maximumFractionDigits > DOUBLE_FRACTION_DIGITS) ?
3547             DOUBLE_FRACTION_DIGITS : maximumFractionDigits);
3548     }
3549     fastPathCheckNeeded = true;
3550 }
3551
3552 /**
3553  * Gets the maximum number of digits allowed in the integer portion of a
3554  * number.
3555  * For formatting numbers other than <code>BigInteger</code> and
3556  * <code>BigDecimal</code> objects, the lower of the return value and
3557  * 309 is used.
3558  * @see #setMaximumIntegerDigits
3559  */
3560 @Override
3561 public int getMaximumIntegerDigits() {
3562     return maximumIntegerDigits;
3563 }
3564
3565 /**
3566  * Gets the minimum number of digits allowed in the integer portion of a
3567  * number.
3568  * For formatting numbers other than <code>BigInteger</code> and
3569  * <code>BigDecimal</code> objects, the lower of the return value and
3570  * 309 is used.
3571  * @see #setMinimumIntegerDigits
3572  */
3573 @Override
3574 public int getMinimumIntegerDigits() {
3575     return minimumIntegerDigits;
3576 }
3577

```

```
3578 /**
3579  * Gets the maximum number of digits allowed in the fraction portion of a
3580  * number.
3581  * For formatting numbers other than BigInteger and
3582  * BigDecimal objects, the lower of the return value and
3583  * 340 is used.
3584  * @see #setMaximumFractionDigits
3585  */
3586 @Override
3587 public int getMaximumFractionDigits() {
3588     return maximumFractionDigits;
3589 }
3590
3591 /**
3592  * Gets the minimum number of digits allowed in the fraction portion of a
3593  * number.
3594  * For formatting numbers other than BigInteger and
3595  * BigDecimal objects, the lower of the return value and
3596  * 340 is used.
3597  * @see #setMinimumFractionDigits
3598  */
3599 @Override
3600 public int getMinimumFractionDigits() {
3601     return minimumFractionDigits;
3602 }
3603
3604 /**
3605  * Gets the currency used by this decimal format when formatting
3606  * currency values.
3607  * The currency is obtained by calling
3608  * {@link DecimalFormatSymbols#getCurrency DecimalFormatSymbols.getCurrency}
3609  * on this number format's symbols.
3610  *
3611  * @return the currency used by this decimal format, or null
3612  * @since 1.4
3613  */
3614 @Override
3615 public Currency getCurrency() {
3616     return symbols.getCurrency();
3617 }
3618
3619 /**
3620  * Sets the currency used by this number format when formatting
3621  * currency values. This does not update the minimum or maximum
3622  * number of fraction digits used by the number format.
3623  * The currency is set by calling
3624  * {@link DecimalFormatSymbols#setCurrency DecimalFormatSymbols.setCurrency}
3625  * on this number format's symbols.
3626  *
3627  * @param currency the new currency to be used by this decimal format
3628  * @exception NullPointerException if currency is null
3629  * @since 1.4
3630  */
```

```

3631     @Override
3632     public void setCurrency(Currency currency) {
3633         if (currency != symbols.getCurrency()) {
3634             symbols.setCurrency(currency);
3635             if (isCurrencyFormat) {
3636                 expandAffixes();
3637             }
3638         }
3639         fastPathCheckNeeded = true;
3640     }
3641
3642     /**
3643      * Gets the {@link java.math.RoundingMode} used in this DecimalFormat.
3644      *
3645      * @return The RoundingMode used for this DecimalFormat.
3646      * @see #setRoundingMode(RoundingMode)
3647      * @since 1.6
3648      */
3649     @Override
3650     public RoundingMode getRoundingMode() {
3651         return roundingMode;
3652     }
3653
3654     /**
3655      * Sets the {@link java.math.RoundingMode} used in this DecimalFormat.
3656      *
3657      * @param roundingMode The RoundingMode to be used
3658      * @see #getRoundingMode()
3659      * @exception NullPointerException if roundingMode is null.
3660      * @since 1.6
3661      */
3662     @Override
3663     public void setRoundingMode(RoundingMode roundingMode) {
3664         if (roundingMode == null) {
3665             throw new NullPointerException();
3666         }
3667
3668         this.roundingMode = roundingMode;
3669         digitList.setRoundingMode(roundingMode);
3670         fastPathCheckNeeded = true;
3671     }
3672
3673     /**
3674      * Reads the default serializable fields from the stream and performs
3675      * validations and adjustments for older serialized versions. The
3676      * validations and adjustments are:
3677      * <ol>
3678      * <li>
3679      * Verify that the superclass's digit count fields correctly reflect
3680      * the limits imposed on formatting numbers other than
3681      * BigInteger and BigDecimal objects. These
3682      * limits are stored in the superclass for serialization compatibility
3683      * with older versions, while the limits for BigInteger and

```

```

3684 * <code>BigDecimal</code> objects are kept in this class.
3685 * If, in the superclass, the minimum or maximum integer digit count is
3686 * larger than <code>DOUBLE_INTEGER_DIGITS</code> or if the minimum or
3687 * maximum fraction digit count is larger than
3688 * <code>DOUBLE_FRACTION_DIGITS</code>, then the stream data is invalid
3689 * and this method throws an <code>InvalidObjectException</code>.
3690 * <li>
3691 * If <code>serialVersionOnStream</code> is less than 4, initialize
3692 * <code>roundingMode</code> to {@link java.math.RoundingMode#HALF_EVEN
3693 * RoundingMode.HALF_EVEN}. This field is new with version 4.
3694 * <li>
3695 * If <code>serialVersionOnStream</code> is less than 3, then call
3696 * the setters for the minimum and maximum integer and fraction digits with
3697 * the values of the corresponding superclass getters to initialize the
3698 * fields in this class. The fields in this class are new with version 3.
3699 * <li>
3700 * If <code>serialVersionOnStream</code> is less than 1, indicating that
3701 * the stream was written by JDK 1.1, initialize
3702 * <code>useExponentialNotation</code>
3703 * to false, since it was not present in JDK 1.1.
3704 * <li>
3705 * Set <code>serialVersionOnStream</code> to the maximum allowed value so
3706 * that default serialization will work properly if this object is streamed
3707 * out again.
3708 * </ol>
3709 *
3710 * <p>Stream versions older than 2 will not have the affix pattern variables
3711 * <code>posPrefixPattern</code> etc. As a result, they will be initialized
3712 * to <code>null</code>, which means the affix strings will be taken as
3713 * literal values. This is exactly what we want, since that corresponds to
3714 * the pre-version-2 behavior.
3715 */
3716 private void readObject(ObjectInputStream stream)
3717     throws IOException, ClassNotFoundException
3718 {
3719     stream.defaultReadObject();
3720     digitList = new DigitList();
3721
3722     // We force complete fast-path reinitialization when the instance is
3723     // deserialized. See clone() comment on fastPathCheckNeeded.
3724     fastPathCheckNeeded = true;
3725     isFastPath = false;
3726     fastPathData = null;
3727
3728     if (serialVersionOnStream < 4) {
3729         setRoundingMode(RoundingMode.HALF_EVEN);
3730     } else {
3731         setRoundingMode(getRoundingMode());
3732     }
3733
3734     // We only need to check the maximum counts because NumberFormat
3735     // .readObject has already ensured that the maximum is greater than the
3736     // minimum count.

```

```

3737     if (super.getMaximumIntegerDigits() > DOUBLE_INTEGER_DIGITS ||
3738         super.getMaximumFractionDigits() > DOUBLE_FRACTION_DIGITS) {
3739         throw new InvalidObjectException("Digit count out of range");
3740     }
3741     if (serialVersionOnStream < 3) {
3742         setMaximumIntegerDigits(super.getMaximumIntegerDigits());
3743         setMinimumIntegerDigits(super.getMinimumIntegerDigits());
3744         setMaximumFractionDigits(super.getMaximumFractionDigits());
3745         setMinimumFractionDigits(super.getMinimumFractionDigits());
3746     }
3747     if (serialVersionOnStream < 1) {
3748         // Didn't have exponential fields
3749         useExponentialNotation = false;
3750     }
3751     serialVersionOnStream = currentSerialVersion;
3752 }
3753
3754 //-----
3755 // INSTANCE VARIABLES
3756 //-----
3757
3758 private transient DigitList digitList = new DigitList();
3759
3760 /**
3761  * The symbol used as a prefix when formatting positive numbers, e.g. "+".
3762  *
3763  * @serial
3764  * @see #getPositivePrefix
3765  */
3766 private String positivePrefix = "";
3767
3768 /**
3769  * The symbol used as a suffix when formatting positive numbers.
3770  * This is often an empty string.
3771  *
3772  * @serial
3773  * @see #getPositiveSuffix
3774  */
3775 private String positiveSuffix = "";
3776
3777 /**
3778  * The symbol used as a prefix when formatting negative numbers, e.g. "-".
3779  *
3780  * @serial
3781  * @see #getNegativePrefix
3782  */
3783 private String negativePrefix = "-";
3784
3785 /**
3786  * The symbol used as a suffix when formatting negative numbers.
3787  * This is often an empty string.
3788  *
3789  * @serial

```



```
3790     * @see #getNegativeSuffix
3791     */
3792     private String negativeSuffix = "";
3793
3794     /**
3795      * The prefix pattern for non-negative numbers. This variable corresponds
3796      * to <code>positivePrefix</code>.
3797      *
3798      * <p>This pattern is expanded by the method <code>expandAffix()</code> to
3799      * <code>positivePrefix</code> to update the latter to reflect changes in
3800      * <code>symbols</code>. If this variable is <code>null</code> then
3801      * <code>positivePrefix</code> is taken as a literal value that does not
3802      * change when <code>symbols</code> changes. This variable is always
3803      * <code>null</code> for <code>DecimalFormat</code> objects older than
3804      * stream version 2 restored from stream.
3805      *
3806      * @serial
3807      * @since 1.3
3808      */
3809     private String posPrefixPattern;
3810
3811     /**
3812      * The suffix pattern for non-negative numbers. This variable corresponds
3813      * to <code>positiveSuffix</code>. This variable is analogous to
3814      * <code>posPrefixPattern</code>; see that variable for further
3815      * documentation.
3816      *
3817      * @serial
3818      * @since 1.3
3819      */
3820     private String posSuffixPattern;
3821
3822     /**
3823      * The prefix pattern for negative numbers. This variable corresponds
3824      * to <code>negativePrefix</code>. This variable is analogous to
3825      * <code>posPrefixPattern</code>; see that variable for further
3826      * documentation.
3827      *
3828      * @serial
3829      * @since 1.3
3830      */
3831     private String negPrefixPattern;
3832
3833     /**
3834      * The suffix pattern for negative numbers. This variable corresponds
3835      * to <code>negativeSuffix</code>. This variable is analogous to
3836      * <code>posPrefixPattern</code>; see that variable for further
3837      * documentation.
3838      *
3839      * @serial
3840      * @since 1.3
3841      */
3842     private String negSuffixPattern;
```

```
3843
3844 /**
3845  * The multiplier for use in percent, per mille, etc.
3846  *
3847  * @serial
3848  * @see #getMultiplier
3849  */
3850 private int    multiplier = 1;
3851
3852 /**
3853  * The number of digits between grouping separators in the integer
3854  * portion of a number. Must be greater than 0 if
3855  * NumberFormat.groupingUsed is true.
3856  *
3857  * @serial
3858  * @see #getGroupingSize
3859  * @see java.text.NumberFormat#isGroupingUsed
3860  */
3861 private byte   groupingSize = 3; // invariant, > 0 if useThousands
3862
3863 /**
3864  * If true, forces the decimal separator to always appear in a formatted
3865  * number, even if the fractional part of the number is zero.
3866  *
3867  * @serial
3868  * @see #isDecimalSeparatorAlwaysShown
3869  */
3870 private boolean decimalSeparatorAlwaysShown = false;
3871
3872 /**
3873  * If true, parse returns BigDecimal wherever possible.
3874  *
3875  * @serial
3876  * @see #isParseBigDecimal
3877  * @since 1.5
3878  */
3879 private boolean parseBigDecimal = false;
3880
3881
3882 /**
3883  * True if this object represents a currency format. This determines
3884  * whether the monetary decimal separator is used instead of the normal one.
3885  */
3886 private transient boolean isCurrencyFormat = false;
3887
3888 /**
3889  * The DecimalFormatSymbols object used by this format.
3890  * It contains the symbols used to format numbers, e.g. the grouping separator,
3891  * decimal separator, and so on.
3892  *
3893  * @serial
3894  * @see #setDecimalFormatSymbols
3895  * @see java.text.DecimalFormatSymbols
```

```

3896     */
3897     private DecimalFormatSymbols symbols = null; // LIU new DecimalFormatSymbols();
3898
3899     /**
3900      * True to force the use of exponential (i.e. scientific) notation when formatting
3901      * numbers.
3902      *
3903      * @serial
3904      * @since 1.2
3905      */
3906     private boolean useExponentialNotation; // Newly persistent in the Java 2 platform v.1.2
3907
3908     /**
3909      * FieldPositions describing the positive prefix String. This is
3910      * lazily created. Use <code>getPositivePrefixFieldPositions</code>
3911      * when needed.
3912      */
3913     private transient FieldPosition[] positivePrefixFieldPositions;
3914
3915     /**
3916      * FieldPositions describing the positive suffix String. This is
3917      * lazily created. Use <code>getPositiveSuffixFieldPositions</code>
3918      * when needed.
3919      */
3920     private transient FieldPosition[] positiveSuffixFieldPositions;
3921
3922     /**
3923      * FieldPositions describing the negative prefix String. This is
3924      * lazily created. Use <code>getNegativePrefixFieldPositions</code>
3925      * when needed.
3926      */
3927     private transient FieldPosition[] negativePrefixFieldPositions;
3928
3929     /**
3930      * FieldPositions describing the negative suffix String. This is
3931      * lazily created. Use <code>getNegativeSuffixFieldPositions</code>
3932      * when needed.
3933      */
3934     private transient FieldPosition[] negativeSuffixFieldPositions;
3935
3936     /**
3937      * The minimum number of digits used to display the exponent when a number is
3938      * formatted in exponential notation. This field is ignored if
3939      * <code>useExponentialNotation</code> is not true.
3940      *
3941      * @serial
3942      * @since 1.2
3943      */
3944     private byte    minExponentDigits; // Newly persistent in the Java 2 platform v.1.2
3945
3946     /**
3947      * The maximum number of digits allowed in the integer portion of a
3948      * <code>BigInteger</code> or <code>BigDecimal</code> number.

```

```
3949 * <code>maximumIntegerDigits</code> must be greater than or equal to
3950 * <code>minimumIntegerDigits</code>.
3951 *
3952 * @serial
3953 * @see #getMaximumIntegerDigits
3954 * @since 1.5
3955 */
3956 private int    maximumIntegerDigits = super.getMaximumIntegerDigits();
3957
3958 /**
3959 * The minimum number of digits allowed in the integer portion of a
3960 * <code>BigInteger</code> or <code>BigDecimal</code> number.
3961 * <code>minimumIntegerDigits</code> must be less than or equal to
3962 * <code>maximumIntegerDigits</code>.
3963 *
3964 * @serial
3965 * @see #getMinimumIntegerDigits
3966 * @since 1.5
3967 */
3968 private int    minimumIntegerDigits = super.getMinimumIntegerDigits();
3969
3970 /**
3971 * The maximum number of digits allowed in the fractional portion of a
3972 * <code>BigInteger</code> or <code>BigDecimal</code> number.
3973 * <code>maximumFractionDigits</code> must be greater than or equal to
3974 * <code>minimumFractionDigits</code>.
3975 *
3976 * @serial
3977 * @see #getMaximumFractionDigits
3978 * @since 1.5
3979 */
3980 private int    maximumFractionDigits = super.getMaximumFractionDigits();
3981
3982 /**
3983 * The minimum number of digits allowed in the fractional portion of a
3984 * <code>BigInteger</code> or <code>BigDecimal</code> number.
3985 * <code>minimumFractionDigits</code> must be less than or equal to
3986 * <code>maximumFractionDigits</code>.
3987 *
3988 * @serial
3989 * @see #getMinimumFractionDigits
3990 * @since 1.5
3991 */
3992 private int    minimumFractionDigits = super.getMinimumFractionDigits();
3993
3994 /**
3995 * The {@link java.math.RoundingMode} used in this DecimalFormat.
3996 *
3997 * @serial
3998 * @since 1.6
3999 */
4000 private RoundingMode roundingMode = RoundingMode.HALF_EVEN;
4001
```

```

4002 // ----- DecimalFormat fields for fast-path for double algorithm -----
4003
4004 /**
4005  * Helper inner utility class for storing the data used in the fast-path
4006  * algorithm. Almost all fields related to fast-path are encapsulated in
4007  * this class.
4008  *
4009  * Any {@code DecimalFormat} instance has a {@code fastPathData}
4010  * reference field that is null unless both the properties of the instance
4011  * are such that the instance is in the "fast-path" state, and a format call
4012  * has been done at least once while in this state.
4013  *
4014  * Almost all fields are related to the "fast-path" state only and don't
4015  * change until one of the instance properties is changed.
4016  *
4017  * {@code firstUsedIndex} and {@code lastFreeIndex} are the only
4018  * two fields that are used and modified while inside a call to
4019  * {@code fastDoubleFormat}.
4020  *
4021  */
4022 private static class FastPathData {
4023     // --- Temporary fields used in fast-path, shared by several methods.
4024
4025     /** The first unused index at the end of the formatted result. */
4026     int lastFreeIndex;
4027
4028     /** The first used index at the beginning of the formatted result */
4029     int firstUsedIndex;
4030
4031     // --- State fields related to fast-path status. Changes due to a
4032     //      property change only. Set by checkAndSetFastPathStatus() only.
4033
4034     /** Difference between locale zero and default zero representation. */
4035     int zeroDelta;
4036
4037     /** Locale char for grouping separator. */
4038     char groupingChar;
4039
4040     /** Fixed index position of last integral digit of formatted result */
4041     int integrallLastIndex;
4042
4043     /** Fixed index position of first fractional digit of formatted result */
4044     int fractionalFirstIndex;
4045
4046     /** Fractional constants depending on decimal|currency state */
4047     double fractionalScaleFactor;
4048     int fractionalMaxIntBound;
4049
4050
4051     /** The char array buffer that will contain the formatted result */
4052     char[] fastPathContainer;
4053
4054     /** Suffixes recorded as char array for efficiency. */

```

```

4055     char[] charsPositivePrefix;
4056     char[] charsNegativePrefix;
4057     char[] charsPositiveSuffix;
4058     char[] charsNegativeSuffix;
4059     boolean positiveAffixesRequired = true;
4060     boolean negativeAffixesRequired = true;
4061 }
4062
4063 /** The format fast-path status of the instance. Logical state. */
4064 private transient boolean isFastPath = false;
4065
4066 /** Flag stating need of check and reinit fast-path status on next format call. */
4067 private transient boolean fastPathCheckNeeded = true;
4068
4069 /** DecimalFormat reference to its FastPathData */
4070 private transient FastPathData fastPathData;
4071
4072
4073 //-----
4074
4075 static final int currentSerialVersion = 4;
4076
4077 /**
4078  * The internal serial version which says which version was written.
4079  * Possible values are:
4080  * <ul>
4081  * <li><b>0</b> (default): versions before the Java 2 platform v1.2
4082  * <li><b>1</b>: version for 1.2, which includes the two new fields
4083  *     <code>useExponentialNotation</code> and
4084  *     <code>minExponentDigits</code>.
4085  * <li><b>2</b>: version for 1.3 and later, which adds four new fields:
4086  *     <code>posPrefixPattern</code>, <code>posSuffixPattern</code>,
4087  *     <code>negPrefixPattern</code>, and <code>negSuffixPattern</code>.
4088  * <li><b>3</b>: version for 1.5 and later, which adds five new fields:
4089  *     <code>maximumIntegerDigits</code>,
4090  *     <code>minimumIntegerDigits</code>,
4091  *     <code>maximumFractionDigits</code>,
4092  *     <code>minimumFractionDigits</code>, and
4093  *     <code>parseBigDecimal</code>.
4094  * <li><b>4</b>: version for 1.6 and later, which adds one new field:
4095  *     <code>roundingMode</code>.
4096  * </ul>
4097  * @since 1.2
4098  * @serial
4099  */
4100 private int serialVersionOnStream = currentSerialVersion;
4101
4102 //-----
4103 // CONSTANTS
4104 //-----
4105
4106 // ----- Fast-Path for double Constants -----
4107

```

```

4108 /** Maximum valid integer value for applying fast-path algorithm */
4109 private static final double MAX_INT_AS_DOUBLE = (double) Integer.MAX_VALUE;
4110
4111 /**
4112  * The digit arrays used in the fast-path methods for collecting digits.
4113  * Using 3 constants arrays of chars ensures a very fast collection of digits
4114  */
4115 private static class DigitArrays {
4116     static final char[] DigitOnes1000 = new char[1000];
4117     static final char[] DigitTens1000 = new char[1000];
4118     static final char[] DigitHundreds1000 = new char[1000];
4119
4120     // initialize on demand holder class idiom for arrays of digits
4121     static {
4122         int tenIndex = 0;
4123         int hundredIndex = 0;
4124         char digitOne = '0';
4125         char digitTen = '0';
4126         char digitHundred = '0';
4127         for (int i = 0; i < 1000; i++) {
4128
4129             DigitOnes1000[i] = digitOne;
4130             if (digitOne == '9')
4131                 digitOne = '0';
4132             else
4133                 digitOne++;
4134
4135             DigitTens1000[i] = digitTen;
4136             if (i == (tenIndex + 9)) {
4137                 tenIndex += 10;
4138                 if (digitTen == '9')
4139                     digitTen = '0';
4140                 else
4141                     digitTen++;
4142             }
4143
4144             DigitHundreds1000[i] = digitHundred;
4145             if (i == (hundredIndex + 99)) {
4146                 digitHundred++;
4147                 hundredIndex += 100;
4148             }
4149         }
4150     }
4151 }
4152 // ----- Fast-Path for double Constants end -----
4153
4154 // Constants for characters used in programmatic (unlocalized) patterns.
4155 private static final char PATTERN_ZERO_DIGIT = '0';
4156 private static final char PATTERN_GROUPING_SEPARATOR = ',';
4157 private static final char PATTERN_DECIMAL_SEPARATOR = '.';
4158 private static final char PATTERN_PER_MILLE = '\u2030';
4159 private static final char PATTERN_PERCENT = '%';
4160 private static final char PATTERN_DIGIT = '#';

```

```

4161 private static final char    PATTERN_SEPARATOR    = ';';
4162 private static final String  PATTERN_EXPONENT    = "E";
4163 private static final char    PATTERN_MINUS      = '-';
4164
4165 /**
4166  * The CURRENCY_SIGN is the standard Unicode symbol for currency. It
4167  * is used in patterns and substituted with either the currency symbol,
4168  * or if it is doubled, with the international currency symbol. If the
4169  * CURRENCY_SIGN is seen in a pattern, then the decimal separator is
4170  * replaced with the monetary decimal separator.
4171  *
4172  * The CURRENCY_SIGN is not localized.
4173  */
4174 private static final char    CURRENCY_SIGN = '\u00A4';
4175
4176 private static final char    QUOTE = '\'';
4177
4178 private static FieldPosition[] EmptyFieldPositionArray = new FieldPosition[0];
4179
4180 // Upper limit on integer and fraction digits for a Java double
4181 static final int DOUBLE_INTEGER_DIGITS = 309;
4182 static final int DOUBLE_FRACTION_DIGITS = 340;
4183
4184 // Upper limit on integer and fraction digits for BigDecimal and BigInteger
4185 static final int MAXIMUM_INTEGER_DIGITS = Integer.MAX_VALUE;
4186 static final int MAXIMUM_FRACTION_DIGITS = Integer.MAX_VALUE;
4187
4188 // Proclaim JDK 1.1 serial compatibility.
4189 static final long serialVersionUID = 864413376551465018L;
4190 }

```

2.16 DecimalFormatSymbols.java

Secuencia 25: java/text/DecimalFormatSymbols.java

```

1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *

```



```
20 *
21 *
22 *
23 *
24 */
25
26 /*
27  * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28  * (C) Copyright IBM Corp. 1996 - 1998 - All Rights Reserved
29  *
30  * The original version of this source code and documentation is copyrighted
31  * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32  * materials are provided under terms of a License Agreement between Taligent
33  * and Sun. This technology is protected by multiple US and International
34  * patents. This notice and attribution to Taligent may not be removed.
35  * Taligent is a registered trademark of Taligent, Inc.
36  *
37  */
38
39 package java.text;
40
41 import java.io.IOException;
42 import java.io.ObjectInputStream;
43 import java.io.Serializable;
44 import java.text.spi.DecimalFormatSymbolsProvider;
45 import java.util.ArrayList;
46 import java.util.Currency;
47 import java.util.List;
48 import java.util.Locale;
49 import java.util.MissingResourceException;
50 import java.util.ResourceBundle;
51 import java.util.concurrent.ConcurrentHashMap;
52 import java.util.concurrent.ConcurrentMap;
53 import sun.util.locale.provider.LocaleProviderAdapter;
54 import sun.util.locale.provider.LocaleServiceProviderPool;
55 import sun.util.locale.provider.ResourceBundleBasedAdapter;
56
57 /**
58  * This class represents the set of symbols (such as the decimal separator,
59  * the grouping separator, and so on) needed by DecimalFormat
60  * to format numbers. DecimalFormat creates for itself an instance of
61  * DecimalFormatSymbols from its locale data. If you need to change any
62  * of these symbols, you can get the DecimalFormatSymbols object from
63  * your DecimalFormat and modify it.
64  *
65  * @see      java.util.Locale
66  * @see      DecimalFormat
67  * @author    Mark Davis
68  * @author    Alan Liu
69  */
70
71 public class DecimalFormatSymbols implements Cloneable, Serializable {
72
```

```

73  /**
74   * Create a DecimalFormatSymbols object for the default
75   * {@link java.util.Locale.Category#FORMAT FORMAT} locale.
76   * This constructor can only construct instances for the locales
77   * supported by the Java runtime environment, not for those
78   * supported by installed
79   * {@link java.text.spi.DecimalFormatSymbolsProvider DecimalFormatSymbolsProvider}
80   * implementations. For full locale coverage, use the
81   * {@link #getInstance(Locale) getInstance} method.
82   * <p>This is equivalent to calling
83   * {@link #DecimalFormatSymbols(Locale)
84   *   DecimalFormatSymbols(Locale.getDefault(Locale.Category.FORMAT))}.
85   * @see java.util.Locale#getDefault(java.util.Locale.Category)
86   * @see java.util.Locale.Category#FORMAT
87   */
88  public DecimalFormatSymbols() {
89      initialize( Locale.getDefault(Locale.Category.FORMAT) );
90  }
91
92  /**
93   * Create a DecimalFormatSymbols object for the given locale.
94   * This constructor can only construct instances for the locales
95   * supported by the Java runtime environment, not for those
96   * supported by installed
97   * {@link java.text.spi.DecimalFormatSymbolsProvider DecimalFormatSymbolsProvider}
98   * implementations. For full locale coverage, use the
99   * {@link #getInstance(Locale) getInstance} method.
100   * If the specified locale contains the {@link java.util.Locale#UNICODE_LOCALE_EXTENSION}
101   * for the numbering system, the instance is initialized with the specified numbering
102   * system if the JRE implementation supports it. For example,
103   * <pre>
104   * NumberFormat.getNumberInstance(Locale.forLanguageTag("th-TH-u-nu-thai"))
105   * </pre>
106   * This may return a {@code NumberFormat} instance with the Thai numbering system,
107   * instead of the Latin numbering system.
108   *
109   * @param locale the desired locale
110   * @exception NullPointerException if <code>locale</code> is null
111   */
112  public DecimalFormatSymbols( Locale locale ) {
113      initialize( locale );
114  }
115
116  /**
117   * Returns an array of all locales for which the
118   * <code>getInstance</code> methods of this class can return
119   * localized instances.
120   * The returned array represents the union of locales supported by the Java
121   * runtime and by installed
122   * {@link java.text.spi.DecimalFormatSymbolsProvider DecimalFormatSymbolsProvider}
123   * implementations. It must contain at least a <code>Locale</code>
124   * instance equal to {@link java.util.Locale#US Locale.US}.
125   *

```

```

126     * @return an array of locales for which localized
127     *         <code>DecimalFormatSymbols</code> instances are available.
128     * @since 1.6
129     */
130     public static Locale[] getAvailableLocales() {
131         LocaleServiceProviderPool pool =
132             LocaleServiceProviderPool.getPool(DecimalFormatSymbolsProvider.class);
133         return pool.getAvailableLocales();
134     }
135
136     /**
137     * Gets the <code>DecimalFormatSymbols</code> instance for the default
138     * locale. This method provides access to <code>DecimalFormatSymbols</code>
139     * instances for locales supported by the Java runtime itself as well
140     * as for those supported by installed
141     * {@link java.text.spi.DecimalFormatSymbolsProvider
142     * DecimalFormatSymbolsProvider} implementations.
143     * <p>This is equivalent to calling
144     * {@link #getInstance(Locale)
145     * getInstance(Locale.getDefault(Locale.Category.FORMAT))}.
146     * @see java.util.Locale#getDefault(java.util.Locale.Category)
147     * @see java.util.Locale.Category#FORMAT
148     * @return a <code>DecimalFormatSymbols</code> instance.
149     * @since 1.6
150     */
151     public static final DecimalFormatSymbols getInstance() {
152         return getInstance(Locale.getDefault(Locale.Category.FORMAT));
153     }
154
155     /**
156     * Gets the <code>DecimalFormatSymbols</code> instance for the specified
157     * locale. This method provides access to <code>DecimalFormatSymbols</code>
158     * instances for locales supported by the Java runtime itself as well
159     * as for those supported by installed
160     * {@link java.text.spi.DecimalFormatSymbolsProvider
161     * DecimalFormatSymbolsProvider} implementations.
162     * If the specified locale contains the {@link java.util.Locale#UNICODE_LOCALE_EXTENSION}
163     * for the numbering system, the instance is initialized with the specified numbering
164     * system if the JRE implementation supports it. For example,
165     * <pre>
166     * NumberFormat.getNumberInstance(Locale.forLanguageTag("th-TH-u-nu-thai"))
167     * </pre>
168     * This may return a {@code NumberFormat} instance with the Thai numbering system,
169     * instead of the Latin numbering system.
170     *
171     * @param locale the desired locale.
172     * @return a <code>DecimalFormatSymbols</code> instance.
173     * @exception NullPointerException if <code>locale</code> is null
174     * @since 1.6
175     */
176     public static final DecimalFormatSymbols getInstance(Locale locale) {
177         LocaleProviderAdapter adapter;
178         adapter = LocaleProviderAdapter.getAdapter(DecimalFormatSymbolsProvider.class, locale);

```

```
179     DecimalFormatSymbolsProvider provider = adapter.getDecimalFormatSymbolsProvider();
180     DecimalFormatSymbols dfsyms = provider.getInstance(locale);
181     if (dfsyms == null) {
182         provider = LocaleProviderAdapter.forJRE().getDecimalFormatSymbolsProvider();
183         dfsyms = provider.getInstance(locale);
184     }
185     return dfsyms;
186 }
187
188 /**
189  * Gets the character used for zero. Different for Arabic, etc.
190  *
191  * @return the character used for zero
192  */
193 public char getZeroDigit() {
194     return zeroDigit;
195 }
196
197 /**
198  * Sets the character used for zero. Different for Arabic, etc.
199  *
200  * @param zeroDigit the character used for zero
201  */
202 public void setZeroDigit(char zeroDigit) {
203     this.zeroDigit = zeroDigit;
204 }
205
206 /**
207  * Gets the character used for thousands separator. Different for French, etc.
208  *
209  * @return the grouping separator
210  */
211 public char getGroupingSeparator() {
212     return groupingSeparator;
213 }
214
215 /**
216  * Sets the character used for thousands separator. Different for French, etc.
217  *
218  * @param groupingSeparator the grouping separator
219  */
220 public void setGroupingSeparator(char groupingSeparator) {
221     this.groupingSeparator = groupingSeparator;
222 }
223
224 /**
225  * Gets the character used for decimal sign. Different for French, etc.
226  *
227  * @return the character used for decimal sign
228  */
229 public char getDecimalSeparator() {
230     return decimalSeparator;
231 }
```

```
232
233 /**
234  * Sets the character used for decimal sign. Different for French, etc.
235  *
236  * @param decimalSeparator the character used for decimal sign
237  */
238 public void setDecimalSeparator(char decimalSeparator) {
239     this.decimalSeparator = decimalSeparator;
240 }
241
242 /**
243  * Gets the character used for per mille sign. Different for Arabic, etc.
244  *
245  * @return the character used for per mille sign
246  */
247 public char getPerMill() {
248     return perMill;
249 }
250
251 /**
252  * Sets the character used for per mille sign. Different for Arabic, etc.
253  *
254  * @param perMill the character used for per mille sign
255  */
256 public void setPerMill(char perMill) {
257     this.perMill = perMill;
258 }
259
260 /**
261  * Gets the character used for percent sign. Different for Arabic, etc.
262  *
263  * @return the character used for percent sign
264  */
265 public char getPercent() {
266     return percent;
267 }
268
269 /**
270  * Sets the character used for percent sign. Different for Arabic, etc.
271  *
272  * @param percent the character used for percent sign
273  */
274 public void setPercent(char percent) {
275     this.percent = percent;
276 }
277
278 /**
279  * Gets the character used for a digit in a pattern.
280  *
281  * @return the character used for a digit in a pattern
282  */
283 public char getDigit() {
284     return digit;
```

```
285     }
286
287     /**
288      * Sets the character used for a digit in a pattern.
289      *
290      * @param digit the character used for a digit in a pattern
291      */
292     public void setDigit(char digit) {
293         this.digit = digit;
294     }
295
296     /**
297      * Gets the character used to separate positive and negative subpatterns
298      * in a pattern.
299      *
300      * @return the pattern separator
301      */
302     public char getPatternSeparator() {
303         return patternSeparator;
304     }
305
306     /**
307      * Sets the character used to separate positive and negative subpatterns
308      * in a pattern.
309      *
310      * @param patternSeparator the pattern separator
311      */
312     public void setPatternSeparator(char patternSeparator) {
313         this.patternSeparator = patternSeparator;
314     }
315
316     /**
317      * Gets the string used to represent infinity. Almost always left
318      * unchanged.
319      *
320      * @return the string representing infinity
321      */
322     public String getInfinity() {
323         return infinity;
324     }
325
326     /**
327      * Sets the string used to represent infinity. Almost always left
328      * unchanged.
329      *
330      * @param infinity the string representing infinity
331      */
332     public void setInfinity(String infinity) {
333         this.infinity = infinity;
334     }
335
336     /**
337      * Gets the string used to represent "not a number". Almost always left
```

```
338     * unchanged.
339     *
340     * @return the string representing "not a number"
341     */
342     public String getNaN() {
343         return NaN;
344     }
345
346     /**
347     * Sets the string used to represent "not a number". Almost always left
348     * unchanged.
349     *
350     * @param NaN the string representing "not a number"
351     */
352     public void setNaN(String NaN) {
353         this.NaN = NaN;
354     }
355
356     /**
357     * Gets the character used to represent minus sign. If no explicit
358     * negative format is specified, one is formed by prefixing
359     * minusSign to the positive format.
360     *
361     * @return the character representing minus sign
362     */
363     public char getMinusSign() {
364         return minusSign;
365     }
366
367     /**
368     * Sets the character used to represent minus sign. If no explicit
369     * negative format is specified, one is formed by prefixing
370     * minusSign to the positive format.
371     *
372     * @param minusSign the character representing minus sign
373     */
374     public void setMinusSign(char minusSign) {
375         this.minusSign = minusSign;
376     }
377
378     /**
379     * Returns the currency symbol for the currency of these
380     * DecimalFormatSymbols in their locale.
381     *
382     * @return the currency symbol
383     * @since 1.2
384     */
385     public String getCurrencySymbol()
386     {
387         return currencySymbol;
388     }
389
390     /**
```

```
391     * Sets the currency symbol for the currency of these
392     * DecimalFormatSymbols in their locale.
393     *
394     * @param currency the currency symbol
395     * @since 1.2
396     */
397     public void setCurrencySymbol(String currency)
398     {
399         currencySymbol = currency;
400     }
401
402     /**
403     * Returns the ISO 4217 currency code of the currency of these
404     * DecimalFormatSymbols.
405     *
406     * @return the currency code
407     * @since 1.2
408     */
409     public String getInternationalCurrencySymbol()
410     {
411         return intlCurrencySymbol;
412     }
413
414     /**
415     * Sets the ISO 4217 currency code of the currency of these
416     * DecimalFormatSymbols.
417     * If the currency code is valid (as defined by
418     * {@link java.util.Currency#getInstance(java.lang.String) Currency.getInstance}),
419     * this also sets the currency attribute to the corresponding Currency
420     * instance and the currency symbol attribute to the currency's symbol
421     * in the DecimalFormatSymbols' locale. If the currency code is not valid,
422     * then the currency attribute is set to null and the currency symbol
423     * attribute is not modified.
424     *
425     * @param currencyCode the currency code
426     * @see #setCurrency
427     * @see #setCurrencySymbol
428     * @since 1.2
429     */
430     public void setInternationalCurrencySymbol(String currencyCode)
431     {
432         intlCurrencySymbol = currencyCode;
433         currency = null;
434         if (currencyCode != null) {
435             try {
436                 currency = Currency.getInstance(currencyCode);
437                 currencySymbol = currency.getSymbol();
438             } catch (IllegalArgumentException e) {
439             }
440         }
441     }
442
443     /**
```



```
444     * Gets the currency of these DecimalFormatSymbols. May be null if the
445     * currency symbol attribute was previously set to a value that's not
446     * a valid ISO 4217 currency code.
447     *
448     * @return the currency used, or null
449     * @since 1.4
450     */
451     public Currency getCurrency() {
452         return currency;
453     }
454
455     /**
456     * Sets the currency of these DecimalFormatSymbols.
457     * This also sets the currency symbol attribute to the currency's symbol
458     * in the DecimalFormatSymbols' locale, and the international currency
459     * symbol attribute to the currency's ISO 4217 currency code.
460     *
461     * @param currency the new currency to be used
462     * @exception NullPointerException if <code>currency</code> is null
463     * @since 1.4
464     * @see #setCurrencySymbol
465     * @see #setInternationalCurrencySymbol
466     */
467     public void setCurrency(Currency currency) {
468         if (currency == null) {
469             throw new NullPointerException();
470         }
471         this.currency = currency;
472         intlCurrencySymbol = currency.getCurrencyCode();
473         currencySymbol = currency.getSymbol(locale);
474     }
475
476
477     /**
478     * Returns the monetary decimal separator.
479     *
480     * @return the monetary decimal separator
481     * @since 1.2
482     */
483     public char getMonetaryDecimalSeparator()
484     {
485         return monetarySeparator;
486     }
487
488     /**
489     * Sets the monetary decimal separator.
490     *
491     * @param sep the monetary decimal separator
492     * @since 1.2
493     */
494     public void setMonetaryDecimalSeparator(char sep)
495     {
496         monetarySeparator = sep;
```

```

497     }
498
499     //-----
500     // BEGIN   Package Private methods ... to be made public later
501     //-----
502
503     /**
504      * Returns the character used to separate the mantissa from the exponent.
505      */
506     char getExponentialSymbol()
507     {
508         return exponential;
509     }
510
511     /**
512      * Returns the string used to separate the mantissa from the exponent.
513      * Examples: "x10^" for 1.23x10^4, "E" for 1.23E4.
514      *
515      * @return the exponent separator string
516      * @see #setExponentSeparator(java.lang.String)
517      * @since 1.6
518      */
519     public String getExponentSeparator()
520     {
521         return exponentialSeparator;
522     }
523
524     /**
525      * Sets the character used to separate the mantissa from the exponent.
526      */
527     void setExponentialSymbol(char exp)
528     {
529         exponential = exp;
530     }
531
532     /**
533      * Sets the string used to separate the mantissa from the exponent.
534      * Examples: "x10^" for 1.23x10^4, "E" for 1.23E4.
535      *
536      * @param exp the exponent separator string
537      * @exception NullPointerException if <code>exp</code> is null
538      * @see #getExponentSeparator()
539      * @since 1.6
540      */
541     public void setExponentSeparator(String exp)
542     {
543         if (exp == null) {
544             throw new NullPointerException();
545         }
546         exponentialSeparator = exp;
547     }
548
549     //-----

```

```

550 // END    Package Private methods ... to be made public later
551 //-----
552
553 /**
554  * Standard override.
555  */
556 @Override
557 public Object clone() {
558     try {
559         return (DecimalFormatSymbols)super.clone();
560         // other fields are bit-copied
561     } catch (CloneNotSupportedException e) {
562         throw new InternalError(e);
563     }
564 }
565
566 /**
567  * Override equals.
568  */
569 @Override
570 public boolean equals(Object obj) {
571     if (obj == null) return false;
572     if (this == obj) return true;
573     if (getClass() != obj.getClass()) return false;
574     DecimalFormatSymbols other = (DecimalFormatSymbols) obj;
575     return (zeroDigit == other.zeroDigit &&
576         groupingSeparator == other.groupingSeparator &&
577         decimalSeparator == other.decimalSeparator &&
578         percent == other.percent &&
579         perMill == other.perMill &&
580         digit == other.digit &&
581         minusSign == other.minusSign &&
582         patternSeparator == other.patternSeparator &&
583         infinity.equals(other.infinity) &&
584         NaN.equals(other.NaN) &&
585         currencySymbol.equals(other.currencySymbol) &&
586         intlCurrencySymbol.equals(other.intlCurrencySymbol) &&
587         currency == other.currency &&
588         monetarySeparator == other.monetarySeparator &&
589         exponentialSeparator.equals(other.exponentialSeparator) &&
590         locale.equals(other.locale));
591 }
592
593 /**
594  * Override hashCode.
595  */
596 @Override
597 public int hashCode() {
598     int result = zeroDigit;
599     result = result * 37 + groupingSeparator;
600     result = result * 37 + decimalSeparator;
601     return result;
602 }

```

```

603
604 /**
605  * Initializes the symbols from the FormatData resource bundle.
606  */
607 private void initialize( Locale locale ) {
608     this.locale = locale;
609
610     // get resource bundle data
611     LocaleProviderAdapter adapter = LocaleProviderAdapter.getAdapter(
612         DecimalFormatSymbolsProvider.class, locale);
613     // Avoid potential recursions
614     if (!(adapter instanceof ResourceBundleBasedAdapter)) {
615         adapter = LocaleProviderAdapter.getResourceBundleBased();
616     }
617     Object[] data = adapter.getLocaleResources(locale).getDecimalFormatSymbolsData();
618     String[] numberElements = (String[]) data[0];
619
620     decimalSeparator = numberElements[0].charAt(0);
621     groupingSeparator = numberElements[1].charAt(0);
622     patternSeparator = numberElements[2].charAt(0);
623     percent = numberElements[3].charAt(0);
624     zeroDigit = numberElements[4].charAt(0); //different for Arabic,etc.
625     digit = numberElements[5].charAt(0);
626     minusSign = numberElements[6].charAt(0);
627     exponential = numberElements[7].charAt(0);
628     exponentialSeparator = numberElements[7]; //string representation new since 1.6
629     perMill = numberElements[8].charAt(0);
630     infinity = numberElements[9];
631     NaN = numberElements[10];
632
633     // Try to obtain the currency used in the locale's country.
634     // Check for empty country string separately because it's a valid
635     // country ID for Locale (and used for the C locale), but not a valid
636     // ISO 3166 country code, and exceptions are expensive.
637     if (locale.getCountry().length() > 0) {
638         try {
639             currency = Currency.getInstance(locale);
640         } catch (IllegalArgumentException e) {
641             // use default values below for compatibility
642         }
643     }
644     if (currency != null) {
645         intlCurrencySymbol = currency.getCurrencyCode();
646         if (data[1] != null && data[1] == intlCurrencySymbol) {
647             currencySymbol = (String) data[2];
648         } else {
649             currencySymbol = currency.getSymbol(locale);
650             data[1] = intlCurrencySymbol;
651             data[2] = currencySymbol;
652         }
653     } else {
654         // default values
655         intlCurrencySymbol = "XXX";

```

```

655         try {
656             currency = Currency.getInstance(intlCurrencySymbol);
657         } catch (IllegalArgumentException e) {
658         }
659         currencySymbol = "\u00A4";
660     }
661     // Currently the monetary decimal separator is the same as the
662     // standard decimal separator for all locales that we support.
663     // If that changes, add a new entry to NumberElements.
664     monetarySeparator = decimalSeparator;
665 }
666
667 /**
668  * Reads the default serializable fields, provides default values for objects
669  * in older serial versions, and initializes non-serializable fields.
670  * If <code>serialVersionOnStream</code>
671  * is less than 1, initializes <code>monetarySeparator</code> to be
672  * the same as <code>decimalSeparator</code> and <code>exponential</code>
673  * to be 'E'.
674  * If <code>serialVersionOnStream</code> is less than 2,
675  * initializes <code>locale</code> to the root locale, and initializes
676  * If <code>serialVersionOnStream</code> is less than 3, it initializes
677  * <code>exponentialSeparator</code> using <code>exponential</code>.
678  * Sets <code>serialVersionOnStream</code> back to the maximum allowed value so that
679  * default serialization will work properly if this object is streamed out again.
680  * Initializes the currency from the intlCurrencySymbol field.
681  *
682  * @since JDK 1.1.6
683  */
684 private void readObject(ObjectInputStream stream)
685     throws IOException, ClassNotFoundException {
686     stream.defaultReadObject();
687     if (serialVersionOnStream < 1) {
688         // Didn't have monetarySeparator or exponential field;
689         // use defaults.
690         monetarySeparator = decimalSeparator;
691         exponential      = 'E';
692     }
693     if (serialVersionOnStream < 2) {
694         // didn't have locale; use root locale
695         locale = Locale.ROOT;
696     }
697     if (serialVersionOnStream < 3) {
698         // didn't have exponentialSeparator. Create one using exponential
699         exponentialSeparator = Character.toString(exponential);
700     }
701     serialVersionOnStream = currentSerialVersion;
702
703     if (intlCurrencySymbol != null) {
704         try {
705             currency = Currency.getInstance(intlCurrencySymbol);
706         } catch (IllegalArgumentException e) {
707         }

```

```
708     }
709 }
710
711 /**
712  * Character used for zero.
713  *
714  * @serial
715  * @see #getZeroDigit
716  */
717 private char    zeroDigit;
718
719 /**
720  * Character used for thousands separator.
721  *
722  * @serial
723  * @see #getGroupingSeparator
724  */
725 private char    groupingSeparator;
726
727 /**
728  * Character used for decimal sign.
729  *
730  * @serial
731  * @see #getDecimalSeparator
732  */
733 private char    decimalSeparator;
734
735 /**
736  * Character used for per mille sign.
737  *
738  * @serial
739  * @see #getPerMill
740  */
741 private char    perMill;
742
743 /**
744  * Character used for percent sign.
745  * @serial
746  * @see #getPercent
747  */
748 private char    percent;
749
750 /**
751  * Character used for a digit in a pattern.
752  *
753  * @serial
754  * @see #getDigit
755  */
756 private char    digit;
757
758 /**
759  * Character used to separate positive and negative subpatterns
760  * in a pattern.
```

```
761     *
762     * @serial
763     * @see #getPatternSeparator
764     */
765     private char    patternSeparator;
766
767     /**
768     * String used to represent infinity.
769     * @serial
770     * @see #getInfinity
771     */
772     private String  infinity;
773
774     /**
775     * String used to represent "not a number".
776     * @serial
777     * @see #getNaN
778     */
779     private String  NaN;
780
781     /**
782     * Character used to represent minus sign.
783     * @serial
784     * @see #getMinusSign
785     */
786     private char    minusSign;
787
788     /**
789     * String denoting the local currency, e.g. "$".
790     * @serial
791     * @see #getCurrencySymbol
792     */
793     private String  currencySymbol;
794
795     /**
796     * ISO 4217 currency code denoting the local currency, e.g. "USD".
797     * @serial
798     * @see #getInternationalCurrencySymbol
799     */
800     private String  intlCurrencySymbol;
801
802     /**
803     * The decimal separator used when formatting currency values.
804     * @serial
805     * @since JDK 1.1.6
806     * @see #getMonetaryDecimalSeparator
807     */
808     private char    monetarySeparator; // Field new in JDK 1.1.6
809
810     /**
811     * The character used to distinguish the exponent in a number formatted
812     * in exponential notation, e.g. 'E' for a number such as "1.23E45".
813     * <p>
```

```

814     * Note that the public API provides no way to set this field,
815     * even though it is supported by the implementation and the stream format.
816     * The intent is that this will be added to the API in the future.
817     *
818     * @serial
819     * @since JDK 1.1.6
820     */
821     private char    exponential;        // Field new in JDK 1.1.6
822
823     /**
824     * The string used to separate the mantissa from the exponent.
825     * Examples: "x10^" for 1.23x10^4, "E" for 1.23E4.
826     * <p>
827     * If both <code>exponential</code> and <code>exponentialSeparator</code>
828     * exist, this <code>exponentialSeparator</code> has the precedence.
829     *
830     * @serial
831     * @since 1.6
832     */
833     private String    exponentialSeparator;    // Field new in JDK 1.6
834
835     /**
836     * The locale of these currency format symbols.
837     *
838     * @serial
839     * @since 1.4
840     */
841     private Locale locale;
842
843     // currency; only the ISO code is serialized.
844     private transient Currency currency;
845
846     // Proclaim JDK 1.1 FCS compatibility
847     static final long serialVersionUID = 5772796243397350300L;
848
849     // The internal serial version which says which version was written
850     // - 0 (default) for version up to JDK 1.1.5
851     // - 1 for version from JDK 1.1.6, which includes two new fields:
852     //     monetarySeparator and exponential.
853     // - 2 for version from J2SE 1.4, which includes locale field.
854     // - 3 for version from J2SE 1.6, which includes exponentialSeparator field.
855     private static final int currentSerialVersion = 3;
856
857     /**
858     * Describes the version of <code>DecimalFormatSymbols</code> present on the stream.
859     * Possible values are:
860     * <ul>
861     * <li><b>0</b> (or uninitialized): versions prior to JDK 1.1.6.
862     *
863     * <li><b>1</b>: Versions written by JDK 1.1.6 or later, which include
864     *     two new fields: <code>monetarySeparator</code> and <code>exponential</code>.
865     * <li><b>2</b>: Versions written by J2SE 1.4 or later, which include a
866     *     new <code>locale</code> field.

```



```

867      * <li><b>3</b></li>: Versions written by J2SE 1.6 or later, which include a
868      *      new <code>exponentialSeparator</code> field.
869      * </ul>
870      * When streaming out a <code>DecimalFormatSymbols</code>, the most recent format
871      * (corresponding to the highest allowable <code>serialVersionOnStream</code>)
872      * is always written.
873      *
874      * @serial
875      * @since JDK 1.1.6
876      */
877     private int serialVersionOnStream = currentSerialVersion;
878 }

```

2.17 DigitList.java

Secuencia 26: java/text/DigitList.java

```

1  /*
2  * Copyright (c) 1996, 2014, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28 * (C) Copyright IBM Corp. 1996 - 1998 - All Rights Reserved
29 *
30 * The original version of this source code and documentation is copyrighted
31 * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32 * materials are provided under terms of a License Agreement between Taligent
33 * and Sun. This technology is protected by multiple US and International
34 * patents. This notice and attribution to Taligent may not be removed.
35 * Taligent is a registered trademark of Taligent, Inc.
36 *
37 */

```

```

38
39 package java.text;
40
41 import java.math.BigDecimal;
42 import java.math.BigInteger;
43 import java.math.RoundingMode;
44 import sun.misc.FloatingDecimal;
45
46 /**
47  * Digit List. Private to DecimalFormat.
48  * Handles the transcoding
49  * between numeric values and strings of characters. Only handles
50  * non-negative numbers. The division of labor between DigitList and
51  * DecimalFormat is that DigitList handles the radix 10 representation
52  * issues; DecimalFormat handles the locale-specific issues such as
53  * positive/negative, grouping, decimal point, currency, and so on.
54  *
55  * A DigitList is really a representation of a floating point value.
56  * It may be an integer value; we assume that a double has sufficient
57  * precision to represent all digits of a long.
58  *
59  * The DigitList representation consists of a string of characters,
60  * which are the digits radix 10, from '0' to '9'. It also has a radix
61  * 10 exponent associated with it. The value represented by a DigitList
62  * object can be computed by multiplying the fraction f, where  $0 \leq f < 1$ ,
63  * derived by placing all the digits of the list to the right of the
64  * decimal point, by  $10^{\text{exponent}}$ .
65  *
66  * @see Locale
67  * @see Format
68  * @see NumberFormat
69  * @see DecimalFormat
70  * @see ChoiceFormat
71  * @see MessageFormat
72  * @author Mark Davis, Alan Liu
73  */
74 final class DigitList implements Cloneable {
75     /**
76      * The maximum number of significant digits in an IEEE 754 double, that
77      * is, in a Java double. This must not be increased, or garbage digits
78      * will be generated, and should not be decreased, or accuracy will be lost.
79      */
80     public static final int MAX_COUNT = 19; // == Long.toString(Long.MAX_VALUE).length()
81
82     /**
83      * These data members are intentionally public and can be set directly.
84      *
85      * The value represented is given by placing the decimal point before
86      * digits[decimalAt]. If decimalAt is < 0, then leading zeros between
87      * the decimal point and the first nonzero digit are implied. If decimalAt
88      * is > count, then trailing zeros between the digits[count-1] and the
89      * decimal point are implied.
90      *

```

```

91     * Equivalently, the represented value is given by  $f * 10^{\text{decimalAt}}$ . Here
92     *  $f$  is a value  $0.1 \leq f < 1$  arrived at by placing the digits in Digits to
93     * the right of the decimal.
94     *
95     * DigitList is normalized, so if it is non-zero, figits[0] is non-zero. We
96     * don't allow denormalized numbers because our exponent is effectively of
97     * unlimited magnitude. The count value contains the number of significant
98     * digits present in digits[].
99     *
100    * Zero is represented by any DigitList with count == 0 or with each digits[i]
101    * for all  $i \leq \text{count} == '0'$ .
102    */
103    public int decimalAt = 0;
104    public int count = 0;
105    public char[] digits = new char[MAX_COUNT];
106
107    private char[] data;
108    private RoundingMode roundingMode = RoundingMode.HALF_EVEN;
109    private boolean isNegative = false;
110
111    /**
112     * Return true if the represented number is zero.
113     */
114    boolean isZero() {
115        for (int i=0; i < count; ++i) {
116            if (digits[i] != '0') {
117                return false;
118            }
119        }
120        return true;
121    }
122
123    /**
124     * Set the rounding mode
125     */
126    void setRoundingMode(RoundingMode r) {
127        roundingMode = r;
128    }
129
130    /**
131     * Clears out the digits.
132     * Use before appending them.
133     * Typically, you set a series of digits with append, then at the point
134     * you hit the decimal point, you set myDigitList.decimalAt = myDigitList.count;
135     * then go on appending digits.
136     */
137    public void clear () {
138        decimalAt = 0;
139        count = 0;
140    }
141
142    /**
143     * Appends a digit to the list, extending the list when necessary.

```

```
144     */
145     public void append(char digit) {
146         if (count == digits.length) {
147             char[] data = new char[count + 100];
148             System.arraycopy(digits, 0, data, 0, count);
149             digits = data;
150         }
151         digits[count++] = digit;
152     }
153
154     /**
155      * Utility routine to get the value of the digit list
156      * If (count == 0) this throws a NumberFormatException, which
157      * mimics Long.parseLong().
158      */
159     public final double getDouble() {
160         if (count == 0) {
161             return 0.0;
162         }
163
164         StringBuffer temp = getStringBuffer();
165         temp.append('.');
166         temp.append(digits, 0, count);
167         temp.append('E');
168         temp.append(decimalAt);
169         return Double.parseDouble(temp.toString());
170     }
171
172     /**
173      * Utility routine to get the value of the digit list.
174      * If (count == 0) this returns 0, unlike Long.parseLong().
175      */
176     public final long getLong() {
177         // for now, simple implementation; later, do proper IEEE native stuff
178
179         if (count == 0) {
180             return 0;
181         }
182
183         // We have to check for this, because this is the one NEGATIVE value
184         // we represent. If we tried to just pass the digits off to parseLong,
185         // we'd get a parse failure.
186         if (isLongMIN_VALUE()) {
187             return Long.MIN_VALUE;
188         }
189
190         StringBuffer temp = getStringBuffer();
191         temp.append(digits, 0, count);
192         for (int i = count; i < decimalAt; ++i) {
193             temp.append('0');
194         }
195         return Long.parseLong(temp.toString());
196     }
```

```

197
198 public final BigDecimal getBigDecimal() {
199     if (count == 0) {
200         if (decimalAt == 0) {
201             return BigDecimal.ZERO;
202         } else {
203             return new BigDecimal("OE" + decimalAt);
204         }
205     }
206
207     if (decimalAt == count) {
208         return new BigDecimal(digits, 0, count);
209     } else {
210         return new BigDecimal(digits, 0, count).scaleByPowerOfTen(decimalAt - count);
211     }
212 }
213
214 /**
215  * Return true if the number represented by this object can fit into
216  * a long.
217  * @param isPositive true if this number should be regarded as positive
218  * @param ignoreNegativeZero true if -0 should be regarded as identical to
219  * +0; otherwise they are considered distinct
220  * @return true if this number fits into a Java long
221  */
222 boolean fitsIntoLong(boolean isPositive, boolean ignoreNegativeZero) {
223     // Figure out if the result will fit in a long. We have to
224     // first look for nonzero digits after the decimal point;
225     // then check the size. If the digit count is 18 or less, then
226     // the value can definitely be represented as a long. If it is 19
227     // then it may be too large.
228
229     // Trim trailing zeros. This does not change the represented value.
230     while (count > 0 && digits[count - 1] == '0') {
231         --count;
232     }
233
234     if (count == 0) {
235         // Positive zero fits into a long, but negative zero can only
236         // be represented as a double. - bug 4162852
237         return isPositive || ignoreNegativeZero;
238     }
239
240     if (decimalAt < count || decimalAt > MAX_COUNT) {
241         return false;
242     }
243
244     if (decimalAt < MAX_COUNT) return true;
245
246     // At this point we have decimalAt == count, and count == MAX_COUNT.
247     // The number will overflow if it is larger than 9223372036854775807
248     // or smaller than -9223372036854775808.
249     for (int i=0; i<count; ++i) {

```

```

250         char dig = digits[i], max = LONG_MIN_REP[i];
251         if (dig > max) return false;
252         if (dig < max) return true;
253     }
254
255     // At this point the first count digits match. If decimalAt is less
256     // than count, then the remaining digits are zero, and we return true.
257     if (count < decimalAt) return true;
258
259     // Now we have a representation of Long.MIN_VALUE, without the leading
260     // negative sign. If this represents a positive value, then it does
261     // not fit; otherwise it fits.
262     return !isPositive;
263 }
264
265 /**
266  * Set the digit list to a representation of the given double value.
267  * This method supports fixed-point notation.
268  * @param isNegative Boolean value indicating whether the number is negative.
269  * @param source Value to be converted; must not be Inf, -Inf, Nan,
270  * or a value <= 0.
271  * @param maximumFractionDigits The most fractional digits which should
272  * be converted.
273  */
274 final void set(boolean isNegative, double source, int maximumFractionDigits) {
275     set(isNegative, source, maximumFractionDigits, true);
276 }
277
278 /**
279  * Set the digit list to a representation of the given double value.
280  * This method supports both fixed-point and exponential notation.
281  * @param isNegative Boolean value indicating whether the number is negative.
282  * @param source Value to be converted; must not be Inf, -Inf, Nan,
283  * or a value <= 0.
284  * @param maximumDigits The most fractional or total digits which should
285  * be converted.
286  * @param fixedPoint If true, then maximumDigits is the maximum
287  * fractional digits to be converted. If false, total digits.
288  */
289 final void set(boolean isNegative, double source, int maximumDigits, boolean fixedPoint) {
290
291     FloatingDecimal.BinaryToASCIIConverter fdConverter = FloatingDecimal.
292         getBinaryToASCIIConverter(source);
293     boolean hasBeenRoundedUp = fdConverter.digitsRoundedUp();
294     boolean valueExactAsDecimal = fdConverter.decimalDigitsExact();
295     assert !fdConverter.isExceptional();
296     String digitsString = fdConverter.toJavaFormatString();
297
298     set(isNegative, digitsString,
299         hasBeenRoundedUp, valueExactAsDecimal,
300         maximumDigits, fixedPoint);
301 }

```

```

302  /**
303   * Generate a representation of the form DDDDD, DDDDD.DDDDD, or
304   * DDDDE+/-DDDD.
305   * @param roundedUp whether or not rounding up has already happened.
306   * @param valueExactAsDecimal whether or not collected digits provide
307   * an exact decimal representation of the value.
308   */
309  private void set(boolean isNegative, String s,
310                  boolean roundedUp, boolean valueExactAsDecimal,
311                  int maximumDigits, boolean fixedPoint) {
312
313      this.isNegative = isNegative;
314      int len = s.length();
315      char[] source = getDataChars(len);
316      s.getChars(0, len, source, 0);
317
318      decimalAt = -1;
319      count = 0;
320      int exponent = 0;
321      // Number of zeros between decimal point and first non-zero digit after
322      // decimal point, for numbers < 1.
323      int leadingZerosAfterDecimal = 0;
324      boolean nonZeroDigitSeen = false;
325
326      for (int i = 0; i < len; ) {
327          char c = source[i++];
328          if (c == '.') {
329              decimalAt = count;
330          } else if (c == 'e' || c == 'E') {
331              exponent = parseInt(source, i, len);
332              break;
333          } else {
334              if (!nonZeroDigitSeen) {
335                  nonZeroDigitSeen = (c != '0');
336                  if (!nonZeroDigitSeen && decimalAt != -1)
337                      ++leadingZerosAfterDecimal;
338              }
339              if (nonZeroDigitSeen) {
340                  digits[count++] = c;
341              }
342          }
343      }
344      if (decimalAt == -1) {
345          decimalAt = count;
346      }
347      if (nonZeroDigitSeen) {
348          decimalAt += exponent - leadingZerosAfterDecimal;
349      }
350
351      if (fixedPoint) {
352          // The negative of the exponent represents the number of leading
353          // zeros between the decimal and the first non-zero digit, for
354          // a value < 0.1 (e.g., for 0.00123, -decimalAt == 2). If this

```

```

355         // is more than the maximum fraction digits, then we have an underflow
356         // for the printed representation.
357         if (-decimalAt > maximumDigits) {
358             // Handle an underflow to zero when we round something like
359             // 0.0009 to 2 fractional digits.
360             count = 0;
361             return;
362         } else if (-decimalAt == maximumDigits) {
363             // If we round 0.0009 to 3 fractional digits, then we have to
364             // create a new one digit in the least significant location.
365             if (shouldRoundUp(0, roundedUp, valueExactAsDecimal)) {
366                 count = 1;
367                 ++decimalAt;
368                 digits[0] = '1';
369             } else {
370                 count = 0;
371             }
372             return;
373         }
374         // else fall through
375     }
376
377     // Eliminate trailing zeros.
378     while (count > 1 && digits[count - 1] == '0') {
379         --count;
380     }
381
382     // Eliminate digits beyond maximum digits to be displayed.
383     // Round up if appropriate.
384     round(fixedPoint ? (maximumDigits + decimalAt) : maximumDigits,
385           roundedUp, valueExactAsDecimal);
386
387 }
388
389 /**
390  * Round the representation to the given number of digits.
391  * @param maximumDigits The maximum number of digits to be shown.
392  * @param alreadyRounded whether or not rounding up has already happened.
393  * @param valueExactAsDecimal whether or not collected digits provide
394  * an exact decimal representation of the value.
395  *
396  * Upon return, count will be less than or equal to maximumDigits.
397  */
398 private final void round(int maximumDigits,
399                          boolean alreadyRounded,
400                          boolean valueExactAsDecimal) {
401     // Eliminate digits beyond maximum digits to be displayed.
402     // Round up if appropriate.
403     if (maximumDigits >= 0 && maximumDigits < count) {
404         if (shouldRoundUp(maximumDigits, alreadyRounded, valueExactAsDecimal)) {
405             // Rounding up involved incrementing digits from LSD to MSD.
406             // In most cases this is simple, but in a worst case situation
407             // (9999..99) we have to adjust the decimalAt value.

```



```

408         for (;;) {
409             --maximumDigits;
410             if (maximumDigits < 0) {
411                 // We have all 9's, so we increment to a single digit
412                 // of one and adjust the exponent.
413                 digits[0] = '1';
414                 ++decimalAt;
415                 maximumDigits = 0; // Adjust the count
416                 break;
417             }
418
419             ++digits[maximumDigits];
420             if (digits[maximumDigits] <= '9') break;
421             // digits[maximumDigits] = '0'; // Unnecessary since we'll truncate this
422         }
423         ++maximumDigits; // Increment for use as count
424     }
425     count = maximumDigits;
426
427     // Eliminate trailing zeros.
428     while (count > 1 && digits[count-1] == '0') {
429         --count;
430     }
431 }
432 }
433
434
435 /**
436  * Return true if truncating the representation to the given number
437  * of digits will result in an increment to the last digit. This
438  * method implements the rounding modes defined in the
439  * java.math.RoundingMode class.
440  * [bnf]
441  * @param maximumDigits the number of digits to keep, from 0 to
442  * <code>count-1</code>. If 0, then all digits are rounded away, and
443  * this method returns true if a one should be generated (e.g., formatting
444  * 0.09 with "#.#").
445  * @param alreadyRounded whether or not rounding up has already happened.
446  * @param valueExactAsDecimal whether or not collected digits provide
447  * an exact decimal representation of the value.
448  * @exception ArithmeticException if rounding is needed with rounding
449  * mode being set to RoundingMode.UNNECESSARY
450  * @return true if digit <code>maximumDigits-1</code> should be
451  * incremented
452  */
453 private boolean shouldRoundUp(int maximumDigits,
454                               boolean alreadyRounded,
455                               boolean valueExactAsDecimal) {
456     if (maximumDigits < count) {
457         /*
458          * To avoid erroneous double-rounding or truncation when converting
459          * a binary double value to text, information about the exactness
460          * of the conversion result in FloatingDecimal, as well as any

```

```

461         * rounding done, is needed in this class.
462         *
463         * - For the HALF_DOWN, HALF_EVEN, HALF_UP rounding rules below:
464         *   In the case of formatting float or double, We must take into
465         *   account what FloatingDecimal has done in the binary to decimal
466         *   conversion.
467         *
468         *   Considering the tie cases, FloatingDecimal may round up the
469         *   value (returning decimal digits equal to tie when it is below),
470         *   or "truncate" the value to the tie while value is above it,
471         *   or provide the exact decimal digits when the binary value can be
472         *   converted exactly to its decimal representation given formatting
473         *   rules of FloatingDecimal ( we have thus an exact decimal
474         *   representation of the binary value).
475         *
476         *   - If the double binary value was converted exactly as a decimal
477         *   value, then DigitList code must apply the expected rounding
478         *   rule.
479         *
480         *   - If FloatingDecimal already rounded up the decimal value,
481         *   DigitList should neither round up the value again in any of
482         *   the three rounding modes above.
483         *
484         *   - If FloatingDecimal has truncated the decimal value to
485         *   an ending '5' digit, DigitList should round up the value in
486         *   all of the three rounding modes above.
487         *
488         *
489         *   This has to be considered only if digit at maximumDigits index
490         *   is exactly the last one in the set of digits, otherwise there are
491         *   remaining digits after that position and we don't have to consider
492         *   what FloatingDecimal did.
493         *
494         * - Other rounding modes are not impacted by these tie cases.
495         *
496         * - For other numbers that are always converted to exact digits
497         *   (like BigInteger, Long, ...), the passed alreadyRounded boolean
498         *   have to be set to false, and valueExactAsDecimal has to be set to
499         *   true in the upper DigitList call stack, providing the right state
500         *   for those situations..
501         */
502
503     switch(roundingMode) {
504     case UP:
505         for (int i=maximumDigits; i<count; ++i) {
506             if (digits[i] != '0') {
507                 return true;
508             }
509         }
510         break;
511     case DOWN:
512         break;
513     case CEILING:

```

```

514         for (int i=maximumDigits; i<count; ++i) {
515             if (digits[i] != '0') {
516                 return !isNegative;
517             }
518         }
519         break;
520     case FLOOR:
521         for (int i=maximumDigits; i<count; ++i) {
522             if (digits[i] != '0') {
523                 return isNegative;
524             }
525         }
526         break;
527     case HALF_UP:
528     case HALF_DOWN:
529         if (digits[maximumDigits] > '5') {
530             // Value is above tie ==> must round up
531             return true;
532         } else if (digits[maximumDigits] == '5') {
533             // Digit at rounding position is a '5'. Tie cases.
534             if (maximumDigits != (count - 1)) {
535                 // There are remaining digits. Above tie => must round up
536                 return true;
537             } else {
538                 // Digit at rounding position is the last one !
539                 if (valueExactAsDecimal) {
540                     // Exact binary representation. On the tie.
541                     // Apply rounding given by roundingMode.
542                     return roundingMode == RoundingMode.HALF_UP;
543                 } else {
544                     // Not an exact binary representation.
545                     // Digit sequence either rounded up or truncated.
546                     // Round up only if it was truncated.
547                     return !alreadyRounded;
548                 }
549             }
550         }
551         // Digit at rounding position is < '5' ==> no round up.
552         // Just let do the default, which is no round up (thus break).
553         break;
554     case HALF_EVEN:
555         // Implement IEEE half-even rounding
556         if (digits[maximumDigits] > '5') {
557             return true;
558         } else if (digits[maximumDigits] == '5') {
559             if (maximumDigits == (count - 1)) {
560                 // the rounding position is exactly the last index :
561                 if (alreadyRounded)
562                     // If FloatingDecimal rounded up (value was below tie),
563                     // then we should not round up again.
564                     return false;
565                 if (!valueExactAsDecimal)

```

```

567         // Otherwise if the digits don't represent exact value,
568         // value was above tie and FloatingDecimal truncated
569         // digits to tie. We must round up.
570         return true;
571     else {
572         // This is an exact tie value, and FloatingDecimal
573         // provided all of the exact digits. We thus apply
574         // HALF_EVEN rounding rule.
575         return ((maximumDigits > 0) &&
576             (digits[maximumDigits-1] % 2 != 0));
577     }
578 } else {
579     // Rounds up if it gives a non null digit after '5'
580     for (int i=maximumDigits+1; i<count; ++i) {
581         if (digits[i] != '0')
582             return true;
583     }
584 }
585 }
586 break;
587 case UNNECESSARY:
588     for (int i=maximumDigits; i<count; ++i) {
589         if (digits[i] != '0') {
590             throw new ArithmeticException(
591                 "Rounding needed with the rounding mode being set to RoundingMode.
592                     UNNECESSARY");
593         }
594     }
595     break;
596 default:
597     assert false;
598 }
599 return false;
600 }
601
602 /**
603  * Utility routine to set the value of the digit list from a long
604  */
605 final void set(boolean isNegative, long source) {
606     set(isNegative, source, 0);
607 }
608
609 /**
610  * Set the digit list to a representation of the given long value.
611  * @param isNegative Boolean value indicating whether the number is negative.
612  * @param source Value to be converted; must be >= 0 or ==
613  * Long.MIN_VALUE.
614  * @param maximumDigits The most digits which should be converted.
615  * If maximumDigits is lower than the number of significant digits
616  * in source, the representation will be rounded. Ignored if <= 0.
617  */
618 final void set(boolean isNegative, long source, int maximumDigits) {

```

```

619         this.isNegative = isNegative;
620
621         // This method does not expect a negative number. However,
622         // "source" can be a Long.MIN_VALUE (-9223372036854775808),
623         // if the number being formatted is a Long.MIN_VALUE. In that
624         // case, it will be formatted as -Long.MIN_VALUE, a number
625         // which is outside the legal range of a long, but which can
626         // be represented by DigitList.
627         if (source <= 0) {
628             if (source == Long.MIN_VALUE) {
629                 decimalAt = count = MAX_COUNT;
630                 System.arraycopy(LONG_MIN_REP, 0, digits, 0, count);
631             } else {
632                 decimalAt = count = 0; // Values <= 0 format as zero
633             }
634         } else {
635             // Rewritten to improve performance. I used to call
636             // Long.toString(), which was about 4x slower than this code.
637             int left = MAX_COUNT;
638             int right;
639             while (source > 0) {
640                 digits[--left] = (char)('0' + (source % 10));
641                 source /= 10;
642             }
643             decimalAt = MAX_COUNT - left;
644             // Don't copy trailing zeros. We are guaranteed that there is at
645             // least one non-zero digit, so we don't have to check lower bounds.
646             for (right = MAX_COUNT - 1; digits[right] == '0'; --right)
647                 ;
648             count = right - left + 1;
649             System.arraycopy(digits, left, digits, 0, count);
650         }
651         if (maximumDigits > 0) round(maximumDigits, false, true);
652     }
653
654     /**
655     * Set the digit list to a representation of the given BigDecimal value.
656     * This method supports both fixed-point and exponential notation.
657     * @param isNegative Boolean value indicating whether the number is negative.
658     * @param source Value to be converted; must not be a value <= 0.
659     * @param maximumDigits The most fractional or total digits which should
660     * be converted.
661     * @param fixedPoint If true, then maximumDigits is the maximum
662     * fractional digits to be converted. If false, total digits.
663     */
664     final void set(boolean isNegative, BigDecimal source, int maximumDigits, boolean fixedPoint) {
665         String s = source.toString();
666         extendDigits(s.length());
667
668         set(isNegative, s,
669             false, true,
670             maximumDigits, fixedPoint);
671     }

```

```

672
673 /**
674  * Set the digit list to a representation of the given BigInteger value.
675  * @param isNegative Boolean value indicating whether the number is negative.
676  * @param source Value to be converted; must be >= 0.
677  * @param maximumDigits The most digits which should be converted.
678  * If maximumDigits is lower than the number of significant digits
679  * in source, the representation will be rounded. Ignored if <= 0.
680  */
681 final void set(boolean isNegative, BigInteger source, int maximumDigits) {
682     this.isNegative = isNegative;
683     String s = source.toString();
684     int len = s.length();
685     extendDigits(len);
686     s.getChars(0, len, digits, 0);
687
688     decimalAt = len;
689     int right;
690     for (right = len - 1; right >= 0 && digits[right] == '0'; --right)
691         ;
692     count = right + 1;
693
694     if (maximumDigits > 0) {
695         round(maximumDigits, false, true);
696     }
697 }
698
699 /**
700  * equality test between two digit lists.
701  */
702 public boolean equals(Object obj) {
703     if (this == obj)                // quick check
704         return true;
705     if (!(obj instanceof DigitList)) // (1) same object?
706         return false;
707     DigitList other = (DigitList) obj;
708     if (count != other.count ||
709         decimalAt != other.decimalAt)
710         return false;
711     for (int i = 0; i < count; i++)
712         if (digits[i] != other.digits[i])
713             return false;
714     return true;
715 }
716
717 /**
718  * Generates the hash code for the digit list.
719  */
720 public int hashCode() {
721     int hashCode = decimalAt;
722
723     for (int i = 0; i < count; i++) {
724         hashCode = hashCode * 37 + digits[i];

```

```

725     }
726
727     return hashCode;
728 }
729
730 /**
731  * Creates a copy of this object.
732  * @return a clone of this instance.
733  */
734 public Object clone() {
735     try {
736         DigitList other = (DigitList) super.clone();
737         char[] newDigits = new char[digits.length];
738         System.arraycopy(digits, 0, newDigits, 0, digits.length);
739         other.digits = newDigits;
740         other.tempBuffer = null;
741         return other;
742     } catch (CloneNotSupportedException e) {
743         throw new InternalError(e);
744     }
745 }
746
747 /**
748  * Returns true if this DigitList represents Long.MIN_VALUE;
749  * false, otherwise. This is required so that getLong() works.
750  */
751 private boolean isLongMIN_VALUE() {
752     if (decimalAt != count || count != MAX_COUNT) {
753         return false;
754     }
755
756     for (int i = 0; i < count; ++i) {
757         if (digits[i] != LONG_MIN_REP[i]) return false;
758     }
759
760     return true;
761 }
762
763 private static final int parseInt(char[] str, int offset, int strLen) {
764     char c;
765     boolean positive = true;
766     if ((c = str[offset]) == '-') {
767         positive = false;
768         offset++;
769     } else if (c == '+') {
770         offset++;
771     }
772
773     int value = 0;
774     while (offset < strLen) {
775         c = str[offset++];
776         if (c >= '0' && c <= '9') {
777             value = value * 10 + (c - '0');

```

```

778         } else {
779             break;
780         }
781     }
782     return positive ? value : -value;
783 }
784
785 // The digit part of -9223372036854775808L
786 private static final char[] LONG_MIN_REP = "9223372036854775808".toCharArray();
787
788 public String toString() {
789     if (isZero()) {
790         return "0";
791     }
792     StringBuffer buf = getStringBuffer();
793     buf.append("0.");
794     buf.append(digits, 0, count);
795     buf.append("x10^");
796     buf.append(decimalAt);
797     return buf.toString();
798 }
799
800 private StringBuffer tempBuffer;
801
802 private StringBuffer getStringBuffer() {
803     if (tempBuffer == null) {
804         tempBuffer = new StringBuffer(MAX_COUNT);
805     } else {
806         tempBuffer.setLength(0);
807     }
808     return tempBuffer;
809 }
810
811 private void extendDigits(int len) {
812     if (len > digits.length) {
813         digits = new char[len];
814     }
815 }
816
817 private final char[] getDataChars(int length) {
818     if (data == null || data.length < length) {
819         data = new char[length];
820     }
821     return data;
822 }
823 }

```

2.18 DontCareFieldPosition.java

Secuencia 27: java/text/DontCareFieldPosition.java

```

1  /*
2  * Copyright (c) 2002, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.

```



```
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 package java.text;
27
28 /**
29  * DontCareFieldPosition defines no-op FieldDelegate. Its
30  * singleton is used for the format methods that don't take a
31  * FieldPosition.
32  */
33 class DontCareFieldPosition extends FieldPosition {
34     // The singleton of DontCareFieldPosition.
35     static final FieldPosition INSTANCE = new DontCareFieldPosition();
36
37     private final Format.FieldDelegate noDelegate = new Format.FieldDelegate() {
38         public void formatted(Format.Field attr, Object value, int start,
39                             int end, StringBuffer buffer) {
40         }
41         public void formatted(int fieldID, Format.Field attr, Object value,
42                             int start, int end, StringBuffer buffer) {
43         }
44     };
45
46     private DontCareFieldPosition() {
47         super(0);
48     }
49
50     Format.FieldDelegate getFieldDelegate() {
51         return noDelegate;
52     }
53 }
```

Secuencia 28: java/text/EntryPair.java

```
1  /*
2  * Copyright (c) 1996, 1998, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright Taligent, Inc. 1996 - All Rights Reserved
28 * (C) Copyright IBM Corp. 1996 - All Rights Reserved
29 *
30 * The original version of this source code and documentation is copyrighted
31 * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32 * materials are provided under terms of a License Agreement between Taligent
33 * and Sun. This technology is protected by multiple US and International
34 * patents. This notice and attribution to Taligent may not be removed.
35 * Taligent is a registered trademark of Taligent, Inc.
36 *
37 */
38
39 package java.text;
40
41 /**
42 * This is used for building contracting character tables. entryName
43 * is the contracting character name and value is its collation
44 * order.
45 */
46 final class EntryPair
47 {
48     public String entryName;
49     public int value;
50     public boolean fwd;
51
52     public EntryPair(String name, int value) {
```

```

53     this(name, value, true);
54 }
55 public EntryPair(String name, int value, boolean fwd) {
56     this.entryName = name;
57     this.value = value;
58     this.fwd = fwd;
59 }
60 }

```

2.20 FieldPosition.java

Secuencia 29: java/text/FieldPosition.java

```

1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright Taligent, Inc. 1996 - All Rights Reserved
28 * (C) Copyright IBM Corp. 1996 - All Rights Reserved
29 *
30 * The original version of this source code and documentation is copyrighted
31 * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32 * materials are provided under terms of a License Agreement between Taligent
33 * and Sun. This technology is protected by multiple US and International
34 * patents. This notice and attribution to Taligent may not be removed.
35 * Taligent is a registered trademark of Taligent, Inc.
36 *
37 */
38
39 package java.text;
40
41 /**

```

```

42 * <code>FieldPosition</code> is a simple class used by <code>Format</code>
43 * and its subclasses to identify fields in formatted output. Fields can
44 * be identified in two ways:
45 * <ul>
46 *   <li>By an integer constant, whose names typically end with
47 *     <code>_FIELD</code>. The constants are defined in the various
48 *     subclasses of <code>Format</code>.
49 *   <li>By a <code>Format.Field</code> constant, see <code>ERA_FIELD</code>
50 *     and its friends in <code>DateFormat</code> for an example.
51 * </ul>
52 * <p>
53 * <code>FieldPosition</code> keeps track of the position of the
54 * field within the formatted output with two indices: the index
55 * of the first character of the field and the index of the last
56 * character of the field.
57 *
58 * <p>
59 * One version of the <code>format</code> method in the various
60 * <code>Format</code> classes requires a <code>FieldPosition</code>
61 * object as an argument. You use this <code>format</code> method
62 * to perform partial formatting or to get information about the
63 * formatted output (such as the position of a field).
64 *
65 * <p>
66 * If you are interested in the positions of all attributes in the
67 * formatted string use the <code>Format</code> method
68 * <code>formatToCharacterIterator</code>.
69 *
70 * @author      Mark Davis
71 * @see         java.text.Format
72 */
73 public class FieldPosition {
74
75     /**
76      * Input: Desired field to determine start and end offsets for.
77      * The meaning depends on the subclass of Format.
78      */
79     int field = 0;
80
81     /**
82      * Output: End offset of field in text.
83      * If the field does not occur in the text, 0 is returned.
84      */
85     int endIndex = 0;
86
87     /**
88      * Output: Start offset of field in text.
89      * If the field does not occur in the text, 0 is returned.
90      */
91     int beginIndex = 0;
92
93     /**
94      * Desired field this FieldPosition is for.

```

```

95     */
96     private Format.Field attribute;
97
98     /**
99     * Creates a FieldPosition object for the given field. Fields are
100    * identified by constants, whose names typically end with _FIELD,
101    * in the various subclasses of Format.
102    *
103    * @param field the field identifier
104    * @see java.text.NumberFormat#INTEGER_FIELD
105    * @see java.text.NumberFormat#FRACTION_FIELD
106    * @see java.text.DateFormat#YEAR_FIELD
107    * @see java.text.DateFormat#MONTH_FIELD
108    */
109     public FieldPosition(int field) {
110         this.field = field;
111     }
112
113     /**
114     * Creates a FieldPosition object for the given field constant. Fields are
115     * identified by constants defined in the various Format
116     * subclasses. This is equivalent to calling
117     * new FieldPosition(attribute, -1).
118     *
119     * @param attribute Format.Field constant identifying a field
120     * @since 1.4
121     */
122     public FieldPosition(Format.Field attribute) {
123         this(attribute, -1);
124     }
125
126     /**
127     * Creates a FieldPosition object for the given field.
128     * The field is identified by an attribute constant from one of the
129     * Field subclasses as well as an integer field ID
130     * defined by the Format subclasses. Format
131     * subclasses that are aware of Field should give precedence
132     * to attribute and ignore fieldID if
133     * attribute is not null. However, older Format
134     * subclasses may not be aware of Field and rely on
135     * fieldID. If the field has no corresponding integer
136     * constant, fieldID should be -1.
137     *
138     * @param attribute Format.Field constant identifying a field
139     * @param fieldID integer constant identifying a field
140     * @since 1.4
141     */
142     public FieldPosition(Format.Field attribute, int fieldID) {
143         this.attribute = attribute;
144         this.field = fieldID;
145     }
146
147     /**

```

```
148     * Returns the field identifier as an attribute constant
149     * from one of the <code>Field</code> subclasses. May return null if
150     * the field is specified only by an integer field ID.
151     *
152     * @return Identifier for the field
153     * @since 1.4
154     */
155     public Format.Field getFieldAttribute() {
156         return attribute;
157     }
158
159     /**
160     * Retrieves the field identifier.
161     *
162     * @return the field identifier
163     */
164     public int getField() {
165         return field;
166     }
167
168     /**
169     * Retrieves the index of the first character in the requested field.
170     *
171     * @return the begin index
172     */
173     public int getBeginIndex() {
174         return beginIndex;
175     }
176
177     /**
178     * Retrieves the index of the character following the last character in the
179     * requested field.
180     *
181     * @return the end index
182     */
183     public int getEndIndex() {
184         return endIndex;
185     }
186
187     /**
188     * Sets the begin index. For use by subclasses of Format.
189     *
190     * @param bi the begin index
191     * @since 1.2
192     */
193     public void setBeginIndex(int bi) {
194         beginIndex = bi;
195     }
196
197     /**
198     * Sets the end index. For use by subclasses of Format.
199     *
200     * @param ei the end index
```

```

201     * @since 1.2
202     */
203     public void setEndIndex(int ei) {
204         endIndex = ei;
205     }
206
207     /**
208     * Returns a Format.FieldDelegate instance that is associated
209     * with the FieldPosition. When the delegate is notified of the same
210     * field the FieldPosition is associated with, the begin/end will be
211     * adjusted.
212     */
213     Format.FieldDelegate getFieldDelegate() {
214         return new Delegate();
215     }
216
217     /**
218     * Overrides equals
219     */
220     public boolean equals(Object obj)
221     {
222         if (obj == null) return false;
223         if (!(obj instanceof FieldPosition))
224             return false;
225         FieldPosition other = (FieldPosition) obj;
226         if (attribute == null) {
227             if (other.attribute != null) {
228                 return false;
229             }
230         }
231         else if (!attribute.equals(other.attribute)) {
232             return false;
233         }
234         return (beginIndex == other.beginIndex
235             && endIndex == other.endIndex
236             && field == other.field);
237     }
238
239     /**
240     * Returns a hash code for this FieldPosition.
241     * @return a hash code value for this object
242     */
243     public int hashCode() {
244         return (field << 24) | (beginIndex << 16) | endIndex;
245     }
246
247     /**
248     * Return a string representation of this FieldPosition.
249     * @return a string representation of this object
250     */
251     public String toString() {
252         return getClass().getName() +
253             "[field=" + field + ",attribute=" + attribute +

```

```

254         ",beginIndex=" + beginIndex +
255         ",endIndex=" + endIndex + ']';
256     }
257
258
259     /**
260      * Return true if the receiver wants a <code>Format.Field</code> value and
261      * <code>attribute</code> is equal to it.
262      */
263     private boolean matchesField(Format.Field attribute) {
264         if (this.attribute != null) {
265             return this.attribute.equals(attribute);
266         }
267         return false;
268     }
269
270     /**
271      * Return true if the receiver wants a <code>Format.Field</code> value and
272      * <code>attribute</code> is equal to it, or true if the receiver
273      * represents an integer constant and <code>field</code> equals it.
274      */
275     private boolean matchesField(Format.Field attribute, int field) {
276         if (this.attribute != null) {
277             return this.attribute.equals(attribute);
278         }
279         return (field == this.field);
280     }
281
282
283     /**
284      * An implementation of FieldDelegate that will adjust the begin/end
285      * of the FieldPosition if the arguments match the field of
286      * the FieldPosition.
287      */
288     private class Delegate implements Format.FieldDelegate {
289         /**
290          * Indicates whether the field has been encountered before. If this
291          * is true, and <code>formatted</code> is invoked, the begin/end
292          * are not updated.
293          */
294         private boolean encounteredField;
295
296         public void formatted(Format.Field attr, Object value, int start,
297                             int end, StringBuffer buffer) {
298             if (!encounteredField && matchesField(attr)) {
299                 setBeginIndex(start);
300                 setEndIndex(end);
301                 encounteredField = (start != end);
302             }
303         }
304
305         public void formatted(int fieldID, Format.Field attr, Object value,
306                             int start, int end, StringBuffer buffer) {

```



```
307         if (!encounteredField && matchesField(attr, fieldID)) {
308             setBeginIndex(start);
309             setEndIndex(end);
310             encounteredField = (start != end);
311         }
312     }
313 }
314 }
```

2.21 Format.java

Secuencia 30: java/text/Format.java

```
1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28 * (C) Copyright IBM Corp. 1996 - 1998 - All Rights Reserved
29 *
30 * The original version of this source code and documentation is copyrighted
31 * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32 * materials are provided under terms of a License Agreement between Taligent
33 * and Sun. This technology is protected by multiple US and International
34 * patents. This notice and attribution to Taligent may not be removed.
35 * Taligent is a registered trademark of Taligent, Inc.
36 *
37 */
38
39 package java.text;
40
41 import java.io.Serializable;
```

```

42
43 /**
44  * <code>Format</code> is an abstract base class for formatting locale-sensitive
45  * information such as dates, messages, and numbers.
46  *
47  * <p>
48  * <code>Format</code> defines the programming interface for formatting
49  * locale-sensitive objects into <code>String</code>s (the
50  * <code>format</code> method) and for parsing <code>String</code>s back
51  * into objects (the <code>parseObject</code> method).
52  *
53  * <p>
54  * Generally, a format's <code>parseObject</code> method must be able to parse
55  * any string formatted by its <code>format</code> method. However, there may
56  * be exceptional cases where this is not possible. For example, a
57  * <code>format</code> method might create two adjacent integer numbers with
58  * no separator in between, and in this case the <code>parseObject</code> could
59  * not tell which digits belong to which number.
60  *
61  * <h3>Subclassing</h3>
62  *
63  * <p>
64  * The Java Platform provides three specialized subclasses of <code>Format</code>--
65  * <code>DateFormat</code>, <code>MessageFormat</code>, and
66  * <code>NumberFormat</code>--for formatting dates, messages, and numbers,
67  * respectively.
68  * <p>
69  * Concrete subclasses must implement three methods:
70  * <ol>
71  * <li> <code>format(Object obj, StringBuffer toAppendTo, FieldPosition pos)</code>
72  * <li> <code>formatToCharacterIterator(Object obj)</code>
73  * <li> <code>parseObject(String source, ParsePosition pos)</code>
74  * </ol>
75  * These general methods allow polymorphic parsing and formatting of objects
76  * and are used, for example, by <code>MessageFormat</code>.
77  * Subclasses often also provide additional <code>format</code> methods for
78  * specific input types as well as <code>parse</code> methods for specific
79  * result types. Any <code>parse</code> method that does not take a
80  * <code>ParsePosition</code> argument should throw <code>ParseException</code>
81  * when no text in the required format is at the beginning of the input text.
82  *
83  * <p>
84  * Most subclasses will also implement the following factory methods:
85  * <ol>
86  * <li>
87  * <code>getInstance</code> for getting a useful format object appropriate
88  * for the current locale
89  * <li>
90  * <code>getInstance(Locale)</code> for getting a useful format
91  * object appropriate for the specified locale
92  * </ol>
93  * In addition, some subclasses may also implement other
94  * <code>getXxxInstance</code> methods for more specialized control. For

```

```

95  * example, the NumberFormat class provides
96  * getPercentInstance and getCurrencyInstance
97  * methods for getting specialized number formatters.
98  *
99  * <p>
100 * Subclasses of Format that allow programmers to create objects
101 * for locales (with getInstance(Locale) for example)
102 * must also implement the following class method:
103 * <blockquote>
104 * <pre>
105 * public static Locale[] getAvailableLocales()
106 * </pre>
107 * </blockquote>
108 *
109 * <p>
110 * And finally subclasses may define a set of constants to identify the various
111 * fields in the formatted output. These constants are used to create a FieldPosition
112 * object which identifies what information is contained in the field and its
113 * position in the formatted result. These constants should be named
114 * <em>item</em>_FIELD where <em>item</em> identifies
115 * the field. For examples of these constants, see ERA_FIELD and its
116 * friends in {@link DateFormat}.
117 *
118 * <h4><a name="synchronization">Synchronization</a></h4>
119 *
120 * <p>
121 * Formats are generally not synchronized.
122 * It is recommended to create separate format instances for each thread.
123 * If multiple threads access a format concurrently, it must be synchronized
124 * externally.
125 *
126 * @see      java.text.ParsePosition
127 * @see      java.text.FieldPosition
128 * @see      java.text.NumberFormat
129 * @see      java.text.DateFormat
130 * @see      java.text.MessageFormat
131 * @author    Mark Davis
132 */
133 public abstract class Format implements Serializable, Cloneable {
134
135     private static final long serialVersionUID = -299282585814624189L;
136
137     /**
138      * Sole constructor. (For invocation by subclass constructors, typically
139      * implicit.)
140      */
141     protected Format() {
142     }
143
144     /**
145      * Formats an object to produce a string. This is equivalent to
146      * <blockquote>
147      * {@link #format\(Object, StringBuffer, FieldPosition\) format}(obj,

```

```

148     *      new StringBuffer(), new FieldPosition(0)).toString();</code>
149     * </blockquote>
150     *
151     * @param obj      The object to format
152     * @return         Formatted string.
153     * @exception IllegalArgumentException if the Format cannot format the given
154     *         object
155     */
156     public final String format (Object obj) {
157         return format(obj, new StringBuffer(), new FieldPosition(0)).toString();
158     }
159
160     /**
161     * Formats an object and appends the resulting text to a given string
162     * buffer.
163     * If the <code>pos</code> argument identifies a field used by the format,
164     * then its indices are set to the beginning and end of the first such
165     * field encountered.
166     *
167     * @param obj      The object to format
168     * @param toAppendTo where the text is to be appended
169     * @param pos      A <code>FieldPosition</code> identifying a field
170     *         in the formatted text
171     * @return         the string buffer passed in as <code>toAppendTo</code>,
172     *         with formatted text appended
173     * @exception NullPointerException if <code>toAppendTo</code> or
174     *         <code>pos</code> is null
175     * @exception IllegalArgumentException if the Format cannot format the given
176     *         object
177     */
178     public abstract StringBuffer format(Object obj,
179                                         StringBuffer toAppendTo,
180                                         FieldPosition pos);
181
182     /**
183     * Formats an Object producing an <code>AttributedCharacterIterator</code>.
184     * You can use the returned <code>AttributedCharacterIterator</code>
185     * to build the resulting String, as well as to determine information
186     * about the resulting String.
187     * <p>
188     * Each attribute key of the AttributedCharacterIterator will be of type
189     * <code>Field</code>. It is up to each <code>Format</code> implementation
190     * to define what the legal values are for each attribute in the
191     * <code>AttributedCharacterIterator</code>, but typically the attribute
192     * key is also used as the attribute value.
193     * <p>The default implementation creates an
194     * <code>AttributedCharacterIterator</code> with no attributes. Subclasses
195     * that support fields should override this and create an
196     * <code>AttributedCharacterIterator</code> with meaningful attributes.
197     *
198     * @exception NullPointerException if obj is null.
199     * @exception IllegalArgumentException when the Format cannot format the
200     *         given object.

```

```

201     * @param obj The object to format
202     * @return AttributedCharacterIterator describing the formatted value.
203     * @since 1.4
204     */
205     public AttributedCharacterIterator formatToCharacterIterator(Object obj) {
206         return createAttributedCharacterIterator(format(obj));
207     }
208
209     /**
210     * Parses text from a string to produce an object.
211     * <p>
212     * The method attempts to parse text starting at the index given by
213     * <code>pos</code>.
214     * If parsing succeeds, then the index of <code>pos</code> is updated
215     * to the index after the last character used (parsing does not necessarily
216     * use all characters up to the end of the string), and the parsed
217     * object is returned. The updated <code>pos</code> can be used to
218     * indicate the starting point for the next call to this method.
219     * If an error occurs, then the index of <code>pos</code> is not
220     * changed, the error index of <code>pos</code> is set to the index of
221     * the character where the error occurred, and null is returned.
222     *
223     * @param source A <code>String</code>, part of which should be parsed.
224     * @param pos A <code>ParsePosition</code> object with index and error
225     *             index information as described above.
226     * @return An <code>Object</code> parsed from the string. In case of
227     *          error, returns null.
228     * @exception NullPointerException if <code>pos</code> is null.
229     */
230     public abstract Object parseObject (String source, ParsePosition pos);
231
232     /**
233     * Parses text from the beginning of the given string to produce an object.
234     * The method may not use the entire text of the given string.
235     *
236     * @param source A <code>String</code> whose beginning should be parsed.
237     * @return An <code>Object</code> parsed from the string.
238     * @exception ParseException if the beginning of the specified string
239     *          cannot be parsed.
240     */
241     public Object parseObject(String source) throws ParseException {
242         ParsePosition pos = new ParsePosition(0);
243         Object result = parseObject(source, pos);
244         if (pos.index == 0) {
245             throw new ParseException("Format.parseObject(String) failed",
246                                     pos.errorIndex);
247         }
248         return result;
249     }
250
251     /**
252     * Creates and returns a copy of this object.
253     *

```

```

254     * @return a clone of this instance.
255     */
256     public Object clone() {
257         try {
258             return super.clone();
259         } catch (CloneNotSupportedException e) {
260             // will never happen
261             throw new InternalError(e);
262         }
263     }
264
265     //
266     // Convenience methods for creating AttributedStringIterators from
267     // different parameters.
268     //
269
270     /**
271     * Creates an <code>AttributedStringIterator</code> for the String
272     * <code>s</code>.
273     *
274     * @param s String to create AttributedStringIterator from
275     * @return AttributedStringIterator wrapping s
276     */
277     AttributedStringIterator createAttributedStringIterator(String s) {
278         AttributedString as = new AttributedString(s);
279
280         return as.getIterator();
281     }
282
283     /**
284     * Creates an <code>AttributedStringIterator</code> containing the
285     * concatenated contents of the passed in
286     * <code>AttributedString</code>s.
287     *
288     * @param iterators AttributedStringIterators used to create resulting
289     *                  AttributedStringIterators
290     * @return AttributedStringIterator wrapping passed in
291     *         AttributedStringIterators
292     */
293     AttributedStringIterator createAttributedStringIterator(
294         AttributedStringIterator[] iterators) {
295         AttributedString as = new AttributedString(iterators);
296
297         return as.getIterator();
298     }
299
300     /**
301     * Returns an AttributedStringIterator with the String
302     * <code>string</code> and additional key/value pair <code>key</code>,
303     * <code>value</code>.
304     *
305     * @param string String to create AttributedStringIterator from
306     * @param key Key for AttributedStringIterator

```

```

307     * @param value Value associated with key in AttributedStringIterator
308     * @return AttributedStringIterator wrapping args
309     */
310     AttributedStringIterator createAttributedString(
311         String string, AttributedStringIterator.Attribute key,
312         Object value) {
313         AttributedString as = new AttributedString(string);
314
315         as.addAttribute(key, value);
316         return as.getIterator();
317     }
318
319     /**
320     * Creates an AttributedStringIterator with the contents of
321     * <code>iterator</code> and the additional attribute <code>key</code>
322     * <code>value</code>.
323     *
324     * @param iterator Initial AttributedStringIterator to add arg to
325     * @param key Key for AttributedStringIterator
326     * @param value Value associated with key in AttributedStringIterator
327     * @return AttributedStringIterator wrapping args
328     */
329     AttributedStringIterator createAttributedString(
330         AttributedStringIterator iterator,
331         AttributedStringIterator.Attribute key, Object value) {
332         AttributedString as = new AttributedString(iterator);
333
334         as.addAttribute(key, value);
335         return as.getIterator();
336     }
337
338
339     /**
340     * Defines constants that are used as attribute keys in the
341     * <code>AttributedString</code> returned
342     * from <code>Format.formatToCharacterIterator</code> and as
343     * field identifiers in <code>FieldPosition</code>.
344     *
345     * @since 1.4
346     */
347     public static class Field extends AttributedStringIterator.Attribute {
348
349         // Proclaim serial compatibility with 1.4 FCS
350         private static final long serialVersionUID = 276966692217360283L;
351
352         /**
353         * Creates a Field with the specified name.
354         *
355         * @param name Name of the attribute
356         */
357         protected Field(String name) {
358             super(name);
359         }

```

```

360     }
361
362
363     /**
364      * FieldDelegate is notified by the various <code>Format</code>
365      * implementations as they are formatting the Objects. This allows for
366      * storage of the individual sections of the formatted String for
367      * later use, such as in a <code>FieldPosition</code> or for an
368      * <code>AttributedCharacterIterator</code>.
369      * <p>
370      * Delegates should NOT assume that the <code>Format</code> will notify
371      * the delegate of fields in any particular order.
372      *
373      * @see FieldPosition#getFieldDelegate
374      * @see CharacterIteratorFieldDelegate
375      */
376     interface FieldDelegate {
377         /**
378          * Notified when a particular region of the String is formatted. This
379          * method will be invoked if there is no corresponding integer field id
380          * matching <code>attr</code>.
381          *
382          * @param attr Identifies the field matched
383          * @param value Value associated with the field
384          * @param start Beginning location of the field, will be >= 0
385          * @param end End of the field, will be >= start and <= buffer.length()
386          * @param buffer Contains current formatted value, receiver should
387          *         NOT modify it.
388          */
389         public void formatted(Format.Field attr, Object value, int start,
390                             int end, StringBuffer buffer);
391
392         /**
393          * Notified when a particular region of the String is formatted.
394          *
395          * @param fieldID Identifies the field by integer
396          * @param attr Identifies the field matched
397          * @param value Value associated with the field
398          * @param start Beginning location of the field, will be >= 0
399          * @param end End of the field, will be >= start and <= buffer.length()
400          * @param buffer Contains current formatted value, receiver should
401          *         NOT modify it.
402          */
403         public void formatted(int fieldID, Format.Field attr, Object value,
404                             int start, int end, StringBuffer buffer);
405     }
406 }

```

2.22 MergeCollation.java

Secuencia 31: java/text/MergeCollation.java

```

1  /*
2  * Copyright (c) 1996, 2012, Oracle and/or its affiliates. All rights reserved.

```



```
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27  * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28  * (C) Copyright IBM Corp. 1996, 1997 - All Rights Reserved
29  *
30  * The original version of this source code and documentation is copyrighted
31  * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32  * materials are provided under terms of a License Agreement between Taligent
33  * and Sun. This technology is protected by multiple US and International
34  * patents. This notice and attribution to Taligent may not be removed.
35  * Taligent is a registered trademark of Taligent, Inc.
36  *
37  */
38
39 package java.text;
40
41 import java.util.ArrayList;
42
43 /**
44  * Utility class for normalizing and merging patterns for collation.
45  * Patterns are strings of the form <entry>*, where <entry> has the
46  * form:
47  * <pattern> := <entry>*
48  * <entry> := <separator><chars>{"/"<extension>}
49  * <separator> := "=", ",", ";", "<", "&"
50  * <chars>, and <extension> are both arbitrary strings.
51  * unquoted whitespaces are ignored.
52  * 'xxx' can be used to quote characters
53  * One difference from Collator is that & is used to reset to a current
54  * point. Or, in other words, it introduces a new sequence which is to
55  * be added to the old.
```

```

56  * That is: "a < b < c < d" is the same as "a < b & b < c & c < d" OR
57  * "a < b < d & b < c"
58  * XXX: make ' ' be a single quote.
59  * @see PatternEntry
60  * @author      Mark Davis, Helena Shih
61  */
62
63  final class MergeCollation {
64
65      /**
66       * Creates from a pattern
67       * @exception ParseException If the input pattern is incorrect.
68       */
69      public MergeCollation(String pattern) throws ParseException
70      {
71          for (int i = 0; i < statusArray.length; i++)
72              statusArray[i] = 0;
73          setPattern(pattern);
74      }
75
76      /**
77       * recovers current pattern
78       */
79      public String getPattern() {
80          return getPattern(true);
81      }
82
83      /**
84       * recovers current pattern.
85       * @param withWhiteSpace puts spacing around the entries, and \n
86       * before & and <
87       */
88      public String getPattern(boolean withWhiteSpace) {
89          StringBuffer result = new StringBuffer();
90          PatternEntry tmp = null;
91          ArrayList<PatternEntry> extList = null;
92          int i;
93          for (i = 0; i < patterns.size(); ++i) {
94              PatternEntry entry = patterns.get(i);
95              if (entry.extension.length() != 0) {
96                  if (extList == null)
97                      extList = new ArrayList<>();
98                  extList.add(entry);
99              } else {
100                  if (extList != null) {
101                      PatternEntry last = findLastWithNoExtension(i-1);
102                      for (int j = extList.size() - 1; j >= 0 ; j--) {
103                          tmp = extList.get(j);
104                          tmp.addToBuffer(result, false, withWhiteSpace, last);
105                      }
106                      extList = null;
107                  }
108                  entry.addToBuffer(result, false, withWhiteSpace, null);

```

```
109     }
110   }
111   if (extList != null) {
112     PatternEntry last = findLastWithNoExtension(i-1);
113     for (int j = extList.size() - 1; j >= 0 ; j--) {
114       tmp = extList.get(j);
115       tmp.addToBuffer(result, false, withWhiteSpace, last);
116     }
117     extList = null;
118   }
119   return result.toString();
120 }
121
122 private final PatternEntry findLastWithNoExtension(int i) {
123   for (--i; i >= 0; --i) {
124     PatternEntry entry = patterns.get(i);
125     if (entry.extension.length() == 0) {
126       return entry;
127     }
128   }
129   return null;
130 }
131
132 /**
133  * emits the pattern for collation builder.
134  * @return emits the string in the format understandable to the collation
135  * builder.
136  */
137 public String emitPattern() {
138   return emitPattern(true);
139 }
140
141 /**
142  * emits the pattern for collation builder.
143  * @param withWhiteSpace puts spacing around the entries, and \n
144  * before & and <
145  * @return emits the string in the format understandable to the collation
146  * builder.
147  */
148 public String emitPattern(boolean withWhiteSpace) {
149   StringBuffer result = new StringBuffer();
150   for (int i = 0; i < patterns.size(); ++i)
151   {
152     PatternEntry entry = patterns.get(i);
153     if (entry != null) {
154       entry.addToBuffer(result, true, withWhiteSpace, null);
155     }
156   }
157   return result.toString();
158 }
159
160 /**
161  * sets the pattern.
```

```

162     */
163     public void setPattern(String pattern) throws ParseException
164     {
165         patterns.clear();
166         addPattern(pattern);
167     }
168
169     /**
170      * adds a pattern to the current one.
171      * @param pattern the new pattern to be added
172      */
173     public void addPattern(String pattern) throws ParseException
174     {
175         if (pattern == null)
176             return;
177
178         PatternEntry.Parser parser = new PatternEntry.Parser(pattern);
179
180         PatternEntry entry = parser.next();
181         while (entry != null) {
182             fixEntry(entry);
183             entry = parser.next();
184         }
185     }
186
187     /**
188      * gets count of separate entries
189      * @return the size of pattern entries
190      */
191     public int getCount() {
192         return patterns.size();
193     }
194
195     /**
196      * gets count of separate entries
197      * @param index the offset of the desired pattern entry
198      * @return the requested pattern entry
199      */
200     public PatternEntry getItemAt(int index) {
201         return patterns.get(index);
202     }
203
204     //=====
205     // privates
206     //=====
207     ArrayList<PatternEntry> patterns = new ArrayList<>(); // a list of PatternEntries
208
209     private transient PatternEntry saveEntry = null;
210     private transient PatternEntry lastEntry = null;
211
212     // This is really used as a local variable inside fixEntry, but we cache
213     // it here to avoid newing it up every time the method is called.
214     private transient StringBuffer excess = new StringBuffer();

```

```

215
216 //
217 // When building a MergeCollation, we need to do lots of searches to see
218 // whether a given entry is already in the table. Since we're using an
219 // array, this would make the algorithm O(N*N). To speed things up, we
220 // use this bit array to remember whether the array contains any entries
221 // starting with each Unicode character. If not, we can avoid the search.
222 // Using BitSet would make this easier, but it's significantly slower.
223 //
224 private transient byte[] statusArray = new byte[8192];
225 private final byte BITARRAYMASK = (byte)0x1;
226 private final int BYTEPOWER = 3;
227 private final int BYTEMASK = (1 << BYTEPOWER) - 1;
228
229 /*
230  * If the strength is RESET, then just change the lastEntry to
231  * be the current. (If the current is not in patterns, signal an error).
232  * If not, then remove the current entry, and add it after lastEntry
233  * (which is usually at the end).
234  */
235 private final void fixEntry(PatternEntry newEntry) throws ParseException
236 {
237     // check to see whether the new entry has the same characters as the previous
238     // entry did (this can happen when a pattern declaring a difference between two
239     // strings that are canonically equivalent is normalized). If so, and the strength
240     // is anything other than IDENTICAL or RESET, throw an exception (you can't
241     // declare a string to be unequal to itself). --rtg 5/24/99
242     if (lastEntry != null && newEntry.chars.equals(lastEntry.chars)
243         && newEntry.extension.equals(lastEntry.extension)) {
244         if (newEntry.strength != Collator.IDENTICAL
245             && newEntry.strength != PatternEntry.RESET) {
246             throw new ParseException("The entries " + lastEntry + " and "
247                 + newEntry + " are adjacent in the rules, but have conflicting "
248                 + "strengths: A character can't be unequal to itself.", -1);
249         } else {
250             // otherwise, just skip this entry and behave as though you never saw it
251             return;
252         }
253     }
254
255     boolean changeLastEntry = true;
256     if (newEntry.strength != PatternEntry.RESET) {
257         int oldIndex = -1;
258
259         if ((newEntry.chars.length() == 1)) {
260
261             char c = newEntry.chars.charAt(0);
262             int statusIndex = c >> BYTEPOWER;
263             byte bitClump = statusArray[statusIndex];
264             byte setBit = (byte)(BITARRAYMASK << (c & BYTEMASK));
265
266             if (bitClump != 0 && (bitClump & setBit) != 0) {
267                 oldIndex = patterns.lastIndexOf(newEntry);

```

```

268         } else {
269             // We're going to add an element that starts with this
270             // character, so go ahead and set its bit.
271             statusArray[statusIndex] = (byte)(bitClump | setBit);
272         }
273     } else {
274         oldIndex = patterns.lastIndexOf(newEntry);
275     }
276     if (oldIndex != -1) {
277         patterns.remove(oldIndex);
278     }
279
280     excess.setLength(0);
281     int lastIndex = findLastEntry(lastEntry, excess);
282
283     if (excess.length() != 0) {
284         newEntry.extension = excess + newEntry.extension;
285         if (lastIndex != patterns.size()) {
286             lastEntry = saveEntry;
287             changeLastEntry = false;
288         }
289     }
290     if (lastIndex == patterns.size()) {
291         patterns.add(newEntry);
292         saveEntry = newEntry;
293     } else {
294         patterns.add(lastIndex, newEntry);
295     }
296 }
297 if (changeLastEntry) {
298     lastEntry = newEntry;
299 }
300 }
301
302 private final int findLastEntry(PatternEntry entry,
303                                 StringBuffer excessChars) throws ParseException
304 {
305     if (entry == null)
306         return 0;
307
308     if (entry.strength != PatternEntry.RESET) {
309         // Search backwards for string that contains this one;
310         // most likely entry is last one
311
312         int oldIndex = -1;
313         if ((entry.chars.length() == 1)) {
314             int index = entry.chars.charAt(0) >> BYTEPOWER;
315             if ((statusArray[index] &
316                 (BITARRAYMASK << (entry.chars.charAt(0) & BYTEMASK))) != 0) {
317                 oldIndex = patterns.lastIndexOf(entry);
318             }
319         } else {
320             oldIndex = patterns.lastIndexOf(entry);

```

```

321     }
322     if ((oldIndex == -1))
323         throw new ParseException("couldn't find last entry: "
324                                 + entry, oldIndex);
325     return oldIndex + 1;
326 } else {
327     int i;
328     for (i = patterns.size() - 1; i >= 0; --i) {
329         PatternEntry e = patterns.get(i);
330         if (e.chars.regionMatches(0, entry.chars, 0,
331                                 e.chars.length())) {
332             excessChars.append(entry.chars.substring(e.chars.length(),
333                                                     entry.chars.length()));
334             break;
335         }
336     }
337     if (i == -1)
338         throw new ParseException("couldn't find: " + entry, i);
339     return i + 1;
340 }
341 }
342 }

```

2.23 MessageFormat.java

Secuencia 32: java/text/MessageFormat.java

```

1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved

```

```
28  * (C) Copyright IBM Corp. 1996 - 1998 - All Rights Reserved
29  *
30  * The original version of this source code and documentation is copyrighted
31  * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32  * materials are provided under terms of a License Agreement between Taligent
33  * and Sun. This technology is protected by multiple US and International
34  * patents. This notice and attribution to Taligent may not be removed.
35  * Taligent is a registered trademark of Taligent, Inc.
36  *
37  */
38
39 package java.text;
40
41 import java.io.InvalidObjectException;
42 import java.io.IOException;
43 import java.io.ObjectInputStream;
44 import java.text.DecimalFormat;
45 import java.util.ArrayList;
46 import java.util.Arrays;
47 import java.util.Date;
48 import java.util.List;
49 import java.util.Locale;
50
51
52 /**
53  * <code>MessageFormat</code> provides a means to produce concatenated
54  * messages in a language-neutral way. Use this to construct messages
55  * displayed for end users.
56  *
57  * <p>
58  * <code>MessageFormat</code> takes a set of objects, formats them, then
59  * inserts the formatted strings into the pattern at the appropriate places.
60  *
61  * <p>
62  * <strong>Note:</strong>
63  * <code>MessageFormat</code> differs from the other <code>Format</code>
64  * classes in that you create a <code>MessageFormat</code> object with one
65  * of its constructors (not with a <code>getInstance</code> style factory
66  * method). The factory methods aren't necessary because <code>MessageFormat</code>
67  * itself doesn't implement locale specific behavior. Any locale specific
68  * behavior is defined by the pattern that you provide as well as the
69  * subformats used for inserted arguments.
70  *
71  * <h3><a name="patterns">Patterns and Their Interpretation</a></h3>
72  *
73  * <code>MessageFormat</code> uses patterns of the following form:
74  * <blockquote><pre>
75  * <i>MessageFormatPattern:</i>
76  *     <i>String</i>
77  *     <i>MessageFormatPattern</i> <i>FormatElement</i> <i>String</i>
78  *
79  * <i>FormatElement:</i>
80  *     { <i>ArgumentIndex</i> }
```



```

81 *      { <i>ArgumentIndex</i> , <i>FormatType</i> }
82 *      { <i>ArgumentIndex</i> , <i>FormatType</i> , <i>FormatStyle</i> }
83 *
84 * <i>FormatType: one of </i>
85 *      number date time choice
86 *
87 * <i>FormatStyle:</i>
88 *      short
89 *      medium
90 *      long
91 *      full
92 *      integer
93 *      currency
94 *      percent
95 *      <i>SubformatPattern</i>
96 * </pre></blockquote>
97 *
98 * <p>Within a <i>String</i>, a pair of single quotes can be used to
99 * quote any arbitrary characters except single quotes. For example,
100 * pattern string <code>"{0}'"</code> represents string
101 * <code>"{0}'"</code>, not a <i>FormatElement</i>. A single quote itself
102 * must be represented by doubled single quotes {<code>'</code> throughout a
103 * <i>String</i>. For example, pattern string <code>"'{''"</code> is
104 * interpreted as a sequence of <code>'</code> (start of quoting and a
105 * left curly brace), <code>'</code> (a single quote), and
106 * <code>}'</code> (a right curly brace and end of quoting),
107 * <em>not</em> <code>'{'</code> and <code>'}'</code> (quoted left and
108 * right curly braces): representing string <code>"{}'"</code>,
109 * <em>not</em> <code>"{}'"</code>.
110 *
111 * <p>A <i>SubformatPattern</i> is interpreted by its corresponding
112 * subformat, and subformat-dependent pattern rules apply. For example,
113 * pattern string <code>"{1,number,<u>$'#,##</u>}"</code>
114 * (<i>SubformatPattern</i> with underline) will produce a number format
115 * with the pound-sign quoted, with a result such as: {<code>
116 * "$#31,45"</code>}. Refer to each {<code>Format</code> subclass documentation for
117 * details.
118 *
119 * <p>Any unmatched quote is treated as closed at the end of the given
120 * pattern. For example, pattern string {<code>"{0}'"</code>} is treated as
121 * pattern {<code>"{0}'"</code>}.
122 *
123 * <p>Any curly braces within an unquoted pattern must be balanced. For
124 * example, <code>"ab {0} de"</code> and <code>"ab {'' de"</code> are
125 * valid patterns, but <code>"ab {0'}' de"</code>, <code>"ab } de"</code>
126 * and <code>""{''"</code> are not.
127 *
128 * <dl><dt><b>Warning:</b></dt><dd>The rules for using quotes within message
129 * format patterns unfortunately have shown to be somewhat confusing.
130 * In particular, it isn't always obvious to localizers whether single
131 * quotes need to be doubled or not. Make sure to inform localizers about
132 * the rules, and tell them (for example, by using comments in resource
133 * bundle source files) which strings will be processed by {<code>MessageFormat</code>}.

```

```

134 * Note that localizers may need to use single quotes in translated
135 * strings where the original version doesn't have them.
136 * </dl>
137 * <p>
138 * The <i>ArgumentIndex</i> value is a non-negative integer written
139 * using the digits {@code '0'} through {@code '9'}, and represents an index into the
140 * {@code arguments} array passed to the {@code format} methods
141 * or the result array returned by the {@code parse} methods.
142 * <p>
143 * The <i>FormatType</i> and <i>FormatStyle</i> values are used to create
144 * a {@code Format} instance for the format element. The following
145 * table shows how the values map to {@code Format} instances. Combinations not
146 * shown in the table are illegal. A <i>SubformatPattern</i> must
147 * be a valid pattern string for the {@code Format} subclass used.
148 *
149 * <table border=1 summary="Shows how FormatType and FormatStyle values map to Format instances">
150 *   <tr>
151 *     <th id="ft" class="TableHeadingColor">FormatType
152 *     <th id="fs" class="TableHeadingColor">FormatStyle
153 *     <th id="sc" class="TableHeadingColor">Subformat Created
154 *   <tr>
155 *     <td headers="ft"><i>(none)</i>
156 *     <td headers="fs"><i>(none)</i>
157 *     <td headers="sc"><code>>null</code>
158 *   <tr>
159 *     <td headers="ft" rowspan=5><code>number</code>
160 *     <td headers="fs"><i>(none)</i>
161 *     <td headers="sc">{@link NumberFormat#getInstance(Locale) NumberFormat.getInstance}{@code (
162 *       getLocale())}
163 *   <tr>
164 *     <td headers="fs"><code>integer</code>
165 *     <td headers="sc">{@link NumberFormat#getIntegerInstance(Locale) NumberFormat.
166 *       getIntegerInstance}{@code (getLocale())}
167 *   <tr>
168 *     <td headers="fs"><code>currency</code>
169 *     <td headers="sc">{@link NumberFormat#getCurrencyInstance(Locale) NumberFormat.
170 *       getCurrencyInstance}{@code (getLocale())}
171 *   <tr>
172 *     <td headers="fs"><code>percent</code>
173 *     <td headers="sc">{@link NumberFormat#getPercentInstance(Locale) NumberFormat.
174 *       getPercentInstance}{@code (getLocale())}
175 *   <tr>
176 *     <td headers="fs"><i>SubformatPattern</i>
177 *     <td headers="sc">{@code new} { @link DecimalFormat#DecimalFormat(String,
178 *       DecimalFormatSymbols) DecimalFormat}{@code (subformatPattern,} { @link DecimalFormatSymbols#
179 *       getInstance(Locale) DecimalFormatSymbols.getInstance}{@code (getLocale())}
180 *   <tr>
181 *     <td headers="ft" rowspan=6><code>date</code>
182 *     <td headers="fs"><i>(none)</i>
183 *     <td headers="sc">{@link DateFormat#getDateInstance(int,Locale) DateFormat.getDateInstance
184 *       }{@code (}{@link DateFormat#DEFAULT}{@code , getLocale())}
185 *   <tr>
186 *     <td headers="fs"><code>short</code>

```

```

180 *      <td headers="sc">{@link DateFormat#getDateInstance(int,Locale) DateFormat.getDateInstance
      }{@code ({@link DateFormat#SHORT}){@code , getLocale()}}
181 *      <tr>
182 *      <td headers="fs"><code>medium</code>
183 *      <td headers="sc">{@link DateFormat#getDateInstance(int,Locale) DateFormat.getDateInstance
      }{@code ({@link DateFormat#DEFAULT}){@code , getLocale()}}
184 *      <tr>
185 *      <td headers="fs"><code>long</code>
186 *      <td headers="sc">{@link DateFormat#getDateInstance(int,Locale) DateFormat.getDateInstance
      }{@code ({@link DateFormat#LONG}){@code , getLocale()}}
187 *      <tr>
188 *      <td headers="fs"><code>full</code>
189 *      <td headers="sc">{@link DateFormat#getDateInstance(int,Locale) DateFormat.getDateInstance
      }{@code ({@link DateFormat#FULL}){@code , getLocale()}}
190 *      <tr>
191 *      <td headers="fs"><i>SubformatPattern</i>
192 *      <td headers="sc">{@code new} {@link SimpleDateFormat#SimpleDateFormat(String,Locale)
      SimpleDateFormat}{@code (subformatPattern, getLocale())}
193 *      <tr>
194 *      <td headers="ft" rowspan=6><code>time</code>
195 *      <td headers="fs"><i>(none)</i>
196 *      <td headers="sc">{@link DateFormat#getTimeInstance(int,Locale) DateFormat.getTimeInstance
      }{@code ({@link DateFormat#DEFAULT}){@code , getLocale()}}
197 *      <tr>
198 *      <td headers="fs"><code>short</code>
199 *      <td headers="sc">{@link DateFormat#getTimeInstance(int,Locale) DateFormat.getTimeInstance
      }{@code ({@link DateFormat#SHORT}){@code , getLocale()}}
200 *      <tr>
201 *      <td headers="fs"><code>medium</code>
202 *      <td headers="sc">{@link DateFormat#getTimeInstance(int,Locale) DateFormat.getTimeInstance
      }{@code ({@link DateFormat#DEFAULT}){@code , getLocale()}}
203 *      <tr>
204 *      <td headers="fs"><code>long</code>
205 *      <td headers="sc">{@link DateFormat#getTimeInstance(int,Locale) DateFormat.getTimeInstance
      }{@code ({@link DateFormat#LONG}){@code , getLocale()}}
206 *      <tr>
207 *      <td headers="fs"><code>full</code>
208 *      <td headers="sc">{@link DateFormat#getTimeInstance(int,Locale) DateFormat.getTimeInstance
      }{@code ({@link DateFormat#FULL}){@code , getLocale()}}
209 *      <tr>
210 *      <td headers="fs"><i>SubformatPattern</i>
211 *      <td headers="sc">{@code new} {@link SimpleDateFormat#SimpleDateFormat(String,Locale)
      SimpleDateFormat}{@code (subformatPattern, getLocale())}
212 *      <tr>
213 *      <td headers="ft"><code>choice</code>
214 *      <td headers="fs"><i>SubformatPattern</i>
215 *      <td headers="sc">{@code new} {@link ChoiceFormat#ChoiceFormat(String) ChoiceFormat}{@code
      (subformatPattern)}
216 * </table>
217 *
218 * <h4>Usage Information</h4>
219 *
220 * <p>

```

```

221 * Here are some examples of usage.
222 * In real internationalized programs, the message format pattern and other
223 * static strings will, of course, be obtained from resource bundles.
224 * Other parameters will be dynamically determined at runtime.
225 * <p>
226 * The first example uses the static method <code>MessageFormat.format</code>,
227 * which internally creates a <code>MessageFormat</code> for one-time use:
228 * <blockquote><pre>
229 * int planet = 7;
230 * String event = "a disturbance in the Force";
231 *
232 * String result = MessageFormat.format(
233 *     "At {1,time} on {1,date}, there was {2} on planet {0,number,integer}.",
234 *     planet, new Date(), event);
235 * </pre></blockquote>
236 * The output is:
237 * <blockquote><pre>
238 * At 12:30 PM on Jul 3, 2053, there was a disturbance in the Force on planet 7.
239 * </pre></blockquote>
240 *
241 * <p>
242 * The following example creates a <code>MessageFormat</code> instance that
243 * can be used repeatedly:
244 * <blockquote><pre>
245 * int fileCount = 1273;
246 * String diskName = "MyDisk";
247 * Object[] testArgs = {new Long(fileCount), diskName};
248 *
249 * MessageFormat form = new MessageFormat(
250 *     "The disk \"{1}\" contains {0} file(s).");
251 *
252 * System.out.println(form.format(testArgs));
253 * </pre></blockquote>
254 * The output with different values for <code>fileCount</code>:
255 * <blockquote><pre>
256 * The disk "MyDisk" contains 0 file(s).
257 * The disk "MyDisk" contains 1 file(s).
258 * The disk "MyDisk" contains 1,273 file(s).
259 * </pre></blockquote>
260 *
261 * <p>
262 * For more sophisticated patterns, you can use a <code>ChoiceFormat</code>
263 * to produce correct forms for singular and plural:
264 * <blockquote><pre>
265 * MessageFormat form = new MessageFormat("The disk \"{1}\" contains {0}.");
266 * double[] filelimits = {0,1,2};
267 * String[] filepart = {"no files","one file","{0,number} files"};
268 * ChoiceFormat fileform = new ChoiceFormat(filelimits, filepart);
269 * form.setFormatByArgumentIndex(0, fileform);
270 *
271 * int fileCount = 1273;
272 * String diskName = "MyDisk";
273 * Object[] testArgs = {new Long(fileCount), diskName};

```

```

274 *
275 * System.out.println(form.format(testArgs));
276 * </pre></blockquote>
277 * The output with different values for <code>fileCount</code>:
278 * <blockquote><pre>
279 * The disk "MyDisk" contains no files.
280 * The disk "MyDisk" contains one file.
281 * The disk "MyDisk" contains 1,273 files.
282 * </pre></blockquote>
283 *
284 * <p>
285 * You can create the <code>ChoiceFormat</code> programmatically, as in the
286 * above example, or by using a pattern. See {@link ChoiceFormat}
287 * for more information.
288 * <blockquote><pre>{@code
289 * form.applyPattern(
290 *     "There {0,choice,0#are no files|1#is one file|1<are {0,number,integer} files}.";
291 * }</pre></blockquote>
292 *
293 * <p>
294 * <strong>Note:</strong> As we see above, the string produced
295 * by a <code>ChoiceFormat</code> in <code>MessageFormat</code> is treated as special;
296 * occurrences of '{' are used to indicate subformats, and cause recursion.
297 * If you create both a <code>MessageFormat</code> and <code>ChoiceFormat</code>
298 * programmatically (instead of using the string patterns), then be careful not to
299 * produce a format that recurses on itself, which will cause an infinite loop.
300 * <p>
301 * When a single argument is parsed more than once in the string, the last match
302 * will be the final result of the parsing. For example,
303 * <blockquote><pre>
304 * MessageFormat mf = new MessageFormat("{0,number,###}, {0,number,##}");
305 * Object[] objs = {new Double(3.1415)};
306 * String result = mf.format( objs );
307 * // result now equals "3.14, 3.1"
308 * objs = null;
309 * objs = mf.parse(result, new ParsePosition(0));
310 * // objs now equals {new Double(3.1)}
311 * </pre></blockquote>
312 *
313 * <p>
314 * Likewise, parsing with a {@code MessageFormat} object using patterns containing
315 * multiple occurrences of the same argument would return the last match. For
316 * example,
317 * <blockquote><pre>
318 * MessageFormat mf = new MessageFormat("{0}, {0}, {0}");
319 * String forParsing = "x, y, z";
320 * Object[] objs = mf.parse(forParsing, new ParsePosition(0));
321 * // result now equals {new String("z")}
322 * </pre></blockquote>
323 *
324 * <h4><a name="synchronization">Synchronization</a></h4>
325 *
326 * <p>

```

```
327 * Message formats are not synchronized.
328 * It is recommended to create separate format instances for each thread.
329 * If multiple threads access a format concurrently, it must be synchronized
330 * externally.
331 *
332 * @see      java.util.Locale
333 * @see      Format
334 * @see      NumberFormat
335 * @see      DecimalFormat
336 * @see      DecimalFormatSymbols
337 * @see      ChoiceFormat
338 * @see      DateFormat
339 * @see      SimpleDateFormat
340 *
341 * @author    Mark Davis
342 */
343
344 public class MessageFormat extends Format {
345
346     private static final long serialVersionUID = 6479157306784022952L;
347
348     /**
349      * Constructs a MessageFormat for the default
350      * {@link java.util.Locale.Category#FORMAT FORMAT} locale and the
351      * specified pattern.
352      * The constructor first sets the locale, then parses the pattern and
353      * creates a list of subformats for the format elements contained in it.
354      * Patterns and their interpretation are specified in the
355      * <a href="#patterns">class description</a>.
356      *
357      * @param pattern the pattern for this message format
358      * @exception IllegalArgumentException if the pattern is invalid
359      */
360     public MessageFormat(String pattern) {
361         this.locale = Locale.getDefault(Locale.Category.FORMAT);
362         applyPattern(pattern);
363     }
364
365     /**
366      * Constructs a MessageFormat for the specified locale and
367      * pattern.
368      * The constructor first sets the locale, then parses the pattern and
369      * creates a list of subformats for the format elements contained in it.
370      * Patterns and their interpretation are specified in the
371      * <a href="#patterns">class description</a>.
372      *
373      * @param pattern the pattern for this message format
374      * @param locale the locale for this message format
375      * @exception IllegalArgumentException if the pattern is invalid
376      * @since 1.4
377      */
378     public MessageFormat(String pattern, Locale locale) {
379         this.locale = locale;
```

```

380     applyPattern(pattern);
381 }
382
383 /**
384  * Sets the locale to be used when creating or comparing subformats.
385  * This affects subsequent calls
386  * <ul>
387  * <li>to the {@link #applyPattern applyPattern}
388  *     and {@link #toPattern toPattern} methods if format elements specify
389  *     a format type and therefore have the subformats created in the
390  *     <code>applyPattern</code> method, as well as
391  * <li>to the <code>format</code> and
392  *     {@link #formatToCharacterIterator formatToCharacterIterator} methods
393  *     if format elements do not specify a format type and therefore have
394  *     the subformats created in the formatting methods.
395  * </ul>
396  * Subformats that have already been created are not affected.
397  *
398  * @param locale the locale to be used when creating or comparing subformats
399  */
400 public void setLocale(Locale locale) {
401     this.locale = locale;
402 }
403
404 /**
405  * Gets the locale that's used when creating or comparing subformats.
406  *
407  * @return the locale used when creating or comparing subformats
408  */
409 public Locale getLocale() {
410     return locale;
411 }
412
413
414 /**
415  * Sets the pattern used by this message format.
416  * The method parses the pattern and creates a list of subformats
417  * for the format elements contained in it.
418  * Patterns and their interpretation are specified in the
419  * <a href="#patterns">class description</a>.
420  *
421  * @param pattern the pattern for this message format
422  * @exception IllegalArgumentException if the pattern is invalid
423  */
424 @SuppressWarnings("fallthrough") // fallthrough in switch is expected, suppress it
425 public void applyPattern(String pattern) {
426     StringBuilder[] segments = new StringBuilder[4];
427     // Allocate only segments[SEG_RAW] here. The rest are
428     // allocated on demand.
429     segments[SEG_RAW] = new StringBuilder();
430
431     int part = SEG_RAW;
432     int formatNumber = 0;

```

```

433     boolean inQuote = false;
434     int braceStack = 0;
435     maxOffset = -1;
436     for (int i = 0; i < pattern.length(); ++i) {
437         char ch = pattern.charAt(i);
438         if (part == SEG_RAW) {
439             if (ch == '\\') {
440                 if (i + 1 < pattern.length()
441                     && pattern.charAt(i+1) == '\\') {
442                     segments[part].append(ch); // handle doubles
443                     ++i;
444                 } else {
445                     inQuote = !inQuote;
446                 }
447             } else if (ch == '{' && !inQuote) {
448                 part = SEG_INDEX;
449                 if (segments[SEG_INDEX] == null) {
450                     segments[SEG_INDEX] = new StringBuilder();
451                 }
452             } else {
453                 segments[part].append(ch);
454             }
455         } else {
456             if (inQuote) { // just copy quotes in parts
457                 segments[part].append(ch);
458                 if (ch == '\\') {
459                     inQuote = false;
460                 }
461             } else {
462                 switch (ch) {
463                     case ',':
464                         if (part < SEG_MODIFIER) {
465                             if (segments[++part] == null) {
466                                 segments[part] = new StringBuilder();
467                             }
468                         } else {
469                             segments[part].append(ch);
470                         }
471                         break;
472                     case '{':
473                         ++braceStack;
474                         segments[part].append(ch);
475                         break;
476                     case '}':
477                         if (braceStack == 0) {
478                             part = SEG_RAW;
479                             makeFormat(i, formatNumber, segments);
480                             formatNumber++;
481                             // throw away other segments
482                             segments[SEG_INDEX] = null;
483                             segments[SEG_TYPE] = null;
484                             segments[SEG_MODIFIER] = null;
485                         } else {

```



```

486         --braceStack;
487         segments[part].append(ch);
488     }
489     break;
490     case ' ':
491         // Skip any leading space chars for SEG_TYPE.
492         if (part != SEG_TYPE || segments[SEG_TYPE].length() > 0) {
493             segments[part].append(ch);
494         }
495         break;
496     case '\\':
497         inQuote = true;
498         // fall through, so we keep quotes in other parts
499     default:
500         segments[part].append(ch);
501         break;
502     }
503 }
504 }
505 }
506 if (braceStack == 0 && part != 0) {
507     maxOffset = -1;
508     throw new IllegalArgumentException("Unmatched braces in the pattern.");
509 }
510 this.pattern = segments[0].toString();
511 }
512
513
514 /**
515  * Returns a pattern representing the current state of the message format.
516  * The string is constructed from internal information and therefore
517  * does not necessarily equal the previously applied pattern.
518  *
519  * @return a pattern representing the current state of the message format
520  */
521 public String toPattern() {
522     // later, make this more extensible
523     int lastOffset = 0;
524     StringBuilder result = new StringBuilder();
525     for (int i = 0; i <= maxOffset; ++i) {
526         copyAndFixQuotes(pattern, lastOffset, offsets[i], result);
527         lastOffset = offsets[i];
528         result.append('{').append(argumentNumbers[i]);
529         Format fmt = formats[i];
530         if (fmt == null) {
531             // do nothing, string format
532         } else if (fmt instanceof NumberFormat) {
533             if (fmt.equals(NumberFormat.getInstance(locale))) {
534                 result.append(",number");
535             } else if (fmt.equals(NumberFormat.getCurrencyInstance(locale))) {
536                 result.append(",number,currency");
537             } else if (fmt.equals(NumberFormat.getPercentInstance(locale))) {
538                 result.append(",number,percent");

```

```

539         } else if (fmt.equals(NumberFormat.getIntegerInstance(locale))) {
540             result.append(",number,integer");
541         } else {
542             if (fmt instanceof DecimalFormat) {
543                 result.append(",number,").append(((DecimalFormat)fmt).toPattern());
544             } else if (fmt instanceof ChoiceFormat) {
545                 result.append(",choice,").append(((ChoiceFormat)fmt).toPattern());
546             } else {
547                 // UNKNOWN
548             }
549         }
550     } else if (fmt instanceof DateFormat) {
551         int index;
552         for (index = MODIFIER_DEFAULT; index < DATE_TIME_MODIFIERS.length; index++) {
553             DateFormat df = DateFormat.getDateInstance(DATE_TIME_MODIFIERS[index],
554                                                         locale);
555             if (fmt.equals(df)) {
556                 result.append(",date");
557                 break;
558             }
559             df = DateFormat.getTimeInstance(DATE_TIME_MODIFIERS[index],
560                                             locale);
561             if (fmt.equals(df)) {
562                 result.append(",time");
563                 break;
564             }
565         }
566         if (index >= DATE_TIME_MODIFIERS.length) {
567             if (fmt instanceof SimpleDateFormat) {
568                 result.append(",date,").append(((SimpleDateFormat)fmt).toPattern());
569             } else {
570                 // UNKNOWN
571             }
572         } else if (index != MODIFIER_DEFAULT) {
573             result.append(',').append(DATE_TIME_MODIFIER_KEYWORDS[index]);
574         }
575     } else {
576         //result.append(", unknown");
577     }
578     result.append('}');
579 }
580 copyAndFixQuotes(pattern, lastOffset, pattern.length(), result);
581 return result.toString();
582 }
583
584 /**
585  * Sets the formats to use for the values passed into
586  * <code>format</code> methods or returned from <code>parse</code>
587  * methods. The indices of elements in <code>newFormats</code>
588  * correspond to the argument indices used in the previously set
589  * pattern string.
590  * The order of formats in <code>newFormats</code> thus corresponds to
591  * the order of elements in the <code>arguments</code> array passed

```

```

592     * to the format methods or the result array returned
593     * by the parse methods.
594     * <p>
595     * If an argument index is used for more than one format element
596     * in the pattern string, then the corresponding new format is used
597     * for all such format elements. If an argument index is not used
598     * for any format element in the pattern string, then the
599     * corresponding new format is ignored. If fewer formats are provided
600     * than needed, then only the formats for argument indices less
601     * than newFormats.length are replaced.
602     *
603     * @param newFormats the new formats to use
604     * @exception NullPointerException if newFormats is null
605     * @since 1.4
606     */
607     public void setFormatsByArgumentIndex(Format[] newFormats) {
608         for (int i = 0; i <= maxOffset; i++) {
609             int j = argumentNumbers[i];
610             if (j < newFormats.length) {
611                 formats[i] = newFormats[j];
612             }
613         }
614     }
615
616     /**
617     * Sets the formats to use for the format elements in the
618     * previously set pattern string.
619     * The order of formats in newFormats corresponds to
620     * the order of format elements in the pattern string.
621     * <p>
622     * If more formats are provided than needed by the pattern string,
623     * the remaining ones are ignored. If fewer formats are provided
624     * than needed, then only the first newFormats.length
625     * formats are replaced.
626     * <p>
627     * Since the order of format elements in a pattern string often
628     * changes during localization, it is generally better to use the
629     * {@link #setFormatsByArgumentIndex setFormatsByArgumentIndex}
630     * method, which assumes an order of formats corresponding to the
631     * order of elements in the arguments array passed to
632     * the format methods or the result array returned by
633     * the parse methods.
634     *
635     * @param newFormats the new formats to use
636     * @exception NullPointerException if newFormats is null
637     */
638     public void setFormats(Format[] newFormats) {
639         int runsToCopy = newFormats.length;
640         if (runsToCopy > maxOffset + 1) {
641             runsToCopy = maxOffset + 1;
642         }
643         for (int i = 0; i < runsToCopy; i++) {
644             formats[i] = newFormats[i];

```

```

645     }
646 }
647
648 /**
649  * Sets the format to use for the format elements within the
650  * previously set pattern string that use the given argument
651  * index.
652  * The argument index is part of the format element definition and
653  * represents an index into the arguments array passed
654  * to the format methods or the result array returned
655  * by the parse methods.
656  * <p>
657  * If the argument index is used for more than one format element
658  * in the pattern string, then the new format is used for all such
659  * format elements. If the argument index is not used for any format
660  * element in the pattern string, then the new format is ignored.
661  *
662  * @param argumentIndex the argument index for which to use the new format
663  * @param newFormat the new format to use
664  * @since 1.4
665  */
666 public void setFormatByArgumentIndex(int argumentIndex, Format newFormat) {
667     for (int j = 0; j <= maxOffset; j++) {
668         if (argumentNumbers[j] == argumentIndex) {
669             formats[j] = newFormat;
670         }
671     }
672 }
673
674 /**
675  * Sets the format to use for the format element with the given
676  * format element index within the previously set pattern string.
677  * The format element index is the zero-based number of the format
678  * element counting from the start of the pattern string.
679  * <p>
680  * Since the order of format elements in a pattern string often
681  * changes during localization, it is generally better to use the
682  * {@link #setFormatByArgumentIndex setFormatByArgumentIndex}
683  * method, which accesses format elements based on the argument
684  * index they specify.
685  *
686  * @param formatElementIndex the index of a format element within the pattern
687  * @param newFormat the format to use for the specified format element
688  * @exception ArrayIndexOutOfBoundsException if {@code formatElementIndex} is equal to or
689  *         larger than the number of format elements in the pattern string
690  */
691 public void setFormat(int formatElementIndex, Format newFormat) {
692     formats[formatElementIndex] = newFormat;
693 }
694
695 /**
696  * Gets the formats used for the values passed into
697  * format methods or returned from parse

```

```

698 * methods. The indices of elements in the returned array
699 * correspond to the argument indices used in the previously set
700 * pattern string.
701 * The order of formats in the returned array thus corresponds to
702 * the order of elements in the <code>arguments</code> array passed
703 * to the <code>format</code> methods or the result array returned
704 * by the <code>parse</code> methods.
705 * <p>
706 * If an argument index is used for more than one format element
707 * in the pattern string, then the format used for the last such
708 * format element is returned in the array. If an argument index
709 * is not used for any format element in the pattern string, then
710 * null is returned in the array.
711 *
712 * @return the formats used for the arguments within the pattern
713 * @since 1.4
714 */
715 public Format[] getFormatsByArgumentIndex() {
716     int maximumArgumentNumber = -1;
717     for (int i = 0; i <= maxOffset; i++) {
718         if (argumentNumbers[i] > maximumArgumentNumber) {
719             maximumArgumentNumber = argumentNumbers[i];
720         }
721     }
722     Format[] resultArray = new Format[maximumArgumentNumber + 1];
723     for (int i = 0; i <= maxOffset; i++) {
724         resultArray[argumentNumbers[i]] = formats[i];
725     }
726     return resultArray;
727 }
728
729 /**
730 * Gets the formats used for the format elements in the
731 * previously set pattern string.
732 * The order of formats in the returned array corresponds to
733 * the order of format elements in the pattern string.
734 * <p>
735 * Since the order of format elements in a pattern string often
736 * changes during localization, it's generally better to use the
737 * {@link #getFormatsByArgumentIndex getFormatsByArgumentIndex}
738 * method, which assumes an order of formats corresponding to the
739 * order of elements in the <code>arguments</code> array passed to
740 * the <code>format</code> methods or the result array returned by
741 * the <code>parse</code> methods.
742 *
743 * @return the formats used for the format elements in the pattern
744 */
745 public Format[] getFormats() {
746     Format[] resultArray = new Format[maxOffset + 1];
747     System.arraycopy(formats, 0, resultArray, 0, maxOffset + 1);
748     return resultArray;
749 }
750

```

```

751 /**
752  * Formats an array of objects and appends the MessageFormat's
753  * pattern, with format elements replaced by the formatted objects, to the
754  * provided StringBuffer.
755  * <p>
756  * The text substituted for the individual format elements is derived from
757  * the current subformat of the format element and the
758  * arguments element at the format element's argument index
759  * as indicated by the first matching line of the following table. An
760  * argument is unavailable if arguments is
761  * null or has fewer than argumentIndex+1 elements.
762  *
763  * <table border=1 summary="Examples of subformat,argument,and formatted text">
764  *   <tr>
765  *     <th>Subformat
766  *     <th>Argument
767  *     <th>Formatted Text
768  *   <tr>
769  *     <td>any</i>
770  *     <td>unavailable</i>
771  *     <td>"{" + argumentIndex + "}"</code>
772  *   <tr>
773  *     <td>any</i>
774  *     <td>null</code>
775  *     <td>"null"</code>
776  *   <tr>
777  *     <td>instanceof ChoiceFormat</code>
778  *     <td>any</i>
779  *     <td>subformat.format(argument).indexOf('{') >= 0 ?<br>
780  *       (new MessageFormat(subformat.format(argument), getLocale()).format(argument) :
781  *       subformat.format(argument)</code>
782  *   <tr>
783  *     <td>!= null</code>
784  *     <td>any</i>
785  *     <td>subformat.format(argument)</code>
786  *   <tr>
787  *     <td>null</code>
788  *     <td>instanceof Number</code>
789  *     <td>NumberFormat.getInstance(getLocale()).format(argument)</code>
790  *   <tr>
791  *     <td>null</code>
792  *     <td>instanceof Date</code>
793  *     <td>DateFormat.getDateTimeInstance(DateFormat.SHORT, DateFormat.SHORT, getLocale
794  *       ()).format(argument)</code>
795  *   <tr>
796  *     <td>null</code>
797  *     <td>instanceof String</code>
798  *     <td>argument</code>
799  *   <tr>
800  *     <td>null</code>
801  *     <td>any</i>
802  *     <td>argument.toString()</code>
803  * </table>

```

```

803 * <p>
804 * If <code>pos</code> is non-null, and refers to
805 * <code>Field.ARGUMENT</code>, the location of the first formatted
806 * string will be returned.
807 *
808 * @param arguments an array of objects to be formatted and substituted.
809 * @param result where text is appended.
810 * @param pos On input: an alignment field, if desired.
811 *           On output: the offsets of the alignment field.
812 * @return the string buffer passed in as {@code result}, with formatted
813 * text appended
814 * @exception IllegalArgumentException if an argument in the
815 *           <code>arguments</code> array is not of the type
816 *           expected by the format element(s) that use it.
817 */
818 public final StringBuffer format(Object[] arguments, StringBuffer result,
819                                 FieldPosition pos)
820 {
821     return subformat(arguments, result, pos, null);
822 }
823
824 /**
825 * Creates a MessageFormat with the given pattern and uses it
826 * to format the given arguments. This is equivalent to
827 * <blockquote>
828 *     <code>(new {@link #MessageFormat(String) MessageFormat}(pattern)).{@link #format(java.
829 *         lang.Object[], java.lang.StringBuffer, java.text.FieldPosition) format}(arguments, new
830 *         StringBuffer(), null).toString())</code>
831 * </blockquote>
832 *
833 * @param pattern the pattern string
834 * @param arguments object(s) to format
835 * @return the formatted string
836 * @exception IllegalArgumentException if the pattern is invalid,
837 *           or if an argument in the <code>arguments</code> array
838 *           is not of the type expected by the format element(s)
839 *           that use it.
840 */
841 public static String format(String pattern, Object ... arguments) {
842     MessageFormat temp = new MessageFormat(pattern);
843     return temp.format(arguments);
844 }
845
846 // Overrides
847 /**
848 * Formats an array of objects and appends the <code>MessageFormat</code>'s
849 * pattern, with format elements replaced by the formatted objects, to the
850 * provided <code>StringBuffer</code>.
851 * This is equivalent to
852 * <blockquote>
853 *     <code>{@link #format(java.lang.Object[], java.lang.StringBuffer, java.text.FieldPosition
854 *         ) format}((Object[]) arguments, result, pos)</code>
855 * </blockquote>

```

```

853     *
854     * @param arguments an array of objects to be formatted and substituted.
855     * @param result where text is appended.
856     * @param pos On input: an alignment field, if desired.
857     *           On output: the offsets of the alignment field.
858     * @exception IllegalArgumentException if an argument in the
859     *           <code>arguments</code> array is not of the type
860     *           expected by the format element(s) that use it.
861     */
862     public final StringBuffer format(Object arguments, StringBuffer result,
863                                     FieldPosition pos)
864     {
865         return subformat((Object[]) arguments, result, pos, null);
866     }
867
868     /**
869     * Formats an array of objects and inserts them into the
870     * <code>MessageFormat</code>'s pattern, producing an
871     * <code>AttributedCharacterIterator</code>.
872     * You can use the returned <code>AttributedCharacterIterator</code>
873     * to build the resulting String, as well as to determine information
874     * about the resulting String.
875     * <p>
876     * The text of the returned <code>AttributedCharacterIterator</code> is
877     * the same that would be returned by
878     * <blockquote>
879     *     <code>{@link #format(java.lang.Object[], java.lang.StringBuffer, java.text.FieldPosition
880     *       ) format}(arguments, new StringBuffer(), null).toString()</code>
881     * </blockquote>
882     * <p>
883     * In addition, the <code>AttributedCharacterIterator</code> contains at
884     * least attributes indicating where text was generated from an
885     * argument in the <code>arguments</code> array. The keys of these attributes are of
886     * type <code>MessageFormat.Field</code>, their values are
887     * <code>Integer</code> objects indicating the index in the <code>arguments</code>
888     * array of the argument from which the text was generated.
889     * <p>
890     * The attributes/value from the underlying <code>Format</code>
891     * instances that <code>MessageFormat</code> uses will also be
892     * placed in the resulting <code>AttributedCharacterIterator</code>.
893     * This allows you to not only find where an argument is placed in the
894     * resulting String, but also which fields it contains in turn.
895     *
896     * @param arguments an array of objects to be formatted and substituted.
897     * @return AttributedCharacterIterator describing the formatted value.
898     * @exception NullPointerException if <code>arguments</code> is null.
899     * @exception IllegalArgumentException if an argument in the
900     *           <code>arguments</code> array is not of the type
901     *           expected by the format element(s) that use it.
902     * @since 1.4
903     */
904     public AttributedCharacterIterator formatToCharacterIterator(Object arguments) {
905         StringBuffer result = new StringBuffer();

```



```

905     ArrayList<AttributedCharacterIterator> iterators = new ArrayList<>();
906
907     if (arguments == null) {
908         throw new NullPointerException(
909             "formatToCharacterIterator must be passed non-null object");
910     }
911     subformat((Object[]) arguments, result, null, iterators);
912     if (iterators.size() == 0) {
913         return createAttributedCharacterIterator("");
914     }
915     return createAttributedCharacterIterator(
916         iterators.toArray(
917             new AttributedCharacterIterator[iterators.size()]));
918 }
919
920 /**
921  * Parses the string.
922  *
923  * <p>Caveats: The parse may fail in a number of circumstances.
924  * For example:
925  * <ul>
926  * <li>If one of the arguments does not occur in the pattern.
927  * <li>If the format of an argument loses information, such as
928  *     with a choice format where a large number formats to "many".
929  * <li>Does not yet handle recursion (where
930  *     the substituted strings contain {n} references.)
931  * <li>Will not always find a match (or the correct match)
932  *     if some part of the parse is ambiguous.
933  *     For example, if the pattern "{1},{2}" is used with the
934  *     string arguments {"a,b", "c"}, it will format as "a,b,c".
935  *     When the result is parsed, it will return {"a", "b,c"}.
936  * <li>If a single argument is parsed more than once in the string,
937  *     then the later parse wins.
938  * </ul>
939  * When the parse fails, use ParsePosition.getErrorIndex() to find out
940  * where in the string the parsing failed. The returned error
941  * index is the starting offset of the sub-patterns that the string
942  * is comparing with. For example, if the parsing string "AAA {0} BBB"
943  * is comparing against the pattern "AAD {0} BBB", the error index is
944  * 0. When an error occurs, the call to this method will return null.
945  * If the source is null, return an empty array.
946  *
947  * @param source the string to parse
948  * @param pos    the parse position
949  * @return an array of parsed objects
950  */
951 public Object[] parse(String source, ParsePosition pos) {
952     if (source == null) {
953         Object[] empty = {};
954         return empty;
955     }
956
957     int maximumArgumentNumber = -1;

```

```

958     for (int i = 0; i <= maxOffset; i++) {
959         if (argumentNumbers[i] > maximumArgumentNumber) {
960             maximumArgumentNumber = argumentNumbers[i];
961         }
962     }
963     Object[] resultArray = new Object[maximumArgumentNumber + 1];
964
965     int patternOffset = 0;
966     int sourceOffset = pos.index;
967     ParsePosition tempStatus = new ParsePosition(0);
968     for (int i = 0; i <= maxOffset; ++i) {
969         // match up to format
970         int len = offsets[i] - patternOffset;
971         if (len == 0 || pattern.regionMatches(patternOffset,
972                                             source, sourceOffset, len)) {
973             sourceOffset += len;
974             patternOffset += len;
975         } else {
976             pos.errorIndex = sourceOffset;
977             return null; // leave index as is to signal error
978         }
979
980         // now use format
981         if (formats[i] == null) { // string format
982             // if at end, use longest possible match
983             // otherwise uses first match to intervening string
984             // does NOT recursively try all possibilities
985             int tempLength = (i != maxOffset) ? offsets[i+1] : pattern.length();
986
987             int next;
988             if (patternOffset >= tempLength) {
989                 next = source.length();
990             } else {
991                 next = source.indexOf(pattern.substring(patternOffset, tempLength),
992                                     sourceOffset);
993             }
994
995             if (next < 0) {
996                 pos.errorIndex = sourceOffset;
997                 return null; // leave index as is to signal error
998             } else {
999                 String strValue= source.substring(sourceOffset,next);
1000                 if (!strValue.equals("{"+argumentNumbers[i]+"}"))
1001                     resultArray[argumentNumbers[i]]
1002                         = source.substring(sourceOffset,next);
1003                 sourceOffset = next;
1004             }
1005         } else {
1006             tempStatus.index = sourceOffset;
1007             resultArray[argumentNumbers[i]]
1008                 = formats[i].parseObject(source,tempStatus);
1009             if (tempStatus.index == sourceOffset) {
1010                 pos.errorIndex = sourceOffset;

```

```

1011         return null; // leave index as is to signal error
1012     }
1013     sourceOffset = tempStatus.index; // update
1014 }
1015 }
1016 int len = pattern.length() - patternOffset;
1017 if (len == 0 || pattern.regionMatches(patternOffset,
1018                                     source, sourceOffset, len)) {
1019     pos.index = sourceOffset + len;
1020 } else {
1021     pos.errorIndex = sourceOffset;
1022     return null; // leave index as is to signal error
1023 }
1024 return resultArray;
1025 }
1026
1027 /**
1028  * Parses text from the beginning of the given string to produce an object
1029  * array.
1030  * The method may not use the entire text of the given string.
1031  * <p>
1032  * See the {@link #parse(String, ParsePosition)} method for more information
1033  * on message parsing.
1034  *
1035  * @param source A <code>String</code> whose beginning should be parsed.
1036  * @return An <code>Object</code> array parsed from the string.
1037  * @exception ParseException if the beginning of the specified string
1038  *         cannot be parsed.
1039  */
1040 public Object[] parse(String source) throws ParseException {
1041     ParsePosition pos = new ParsePosition(0);
1042     Object[] result = parse(source, pos);
1043     if (pos.index == 0) // unchanged, returned object is null
1044         throw new ParseException("MessageFormat parse error!", pos.errorIndex);
1045
1046     return result;
1047 }
1048
1049 /**
1050  * Parses text from a string to produce an object array.
1051  * <p>
1052  * The method attempts to parse text starting at the index given by
1053  * <code>pos</code>.
1054  * If parsing succeeds, then the index of <code>pos</code> is updated
1055  * to the index after the last character used (parsing does not necessarily
1056  * use all characters up to the end of the string), and the parsed
1057  * object array is returned. The updated <code>pos</code> can be used to
1058  * indicate the starting point for the next call to this method.
1059  * If an error occurs, then the index of <code>pos</code> is not
1060  * changed, the error index of <code>pos</code> is set to the index of
1061  * the character where the error occurred, and null is returned.
1062  * <p>
1063  * See the {@link #parse(String, ParsePosition)} method for more information

```

```

1064     * on message parsing.
1065     *
1066     * @param source A <code>String</code>, part of which should be parsed.
1067     * @param pos A <code>ParsePosition</code> object with index and error
1068     *           index information as described above.
1069     * @return An <code>Object</code> array parsed from the string. In case of
1070     *         error, returns null.
1071     * @exception NullPointerException if <code>pos</code> is null.
1072     */
1073     public Object parseObject(String source, ParsePosition pos) {
1074         return parse(source, pos);
1075     }
1076
1077     /**
1078     * Creates and returns a copy of this object.
1079     *
1080     * @return a clone of this instance.
1081     */
1082     public Object clone() {
1083         MessageFormat other = (MessageFormat) super.clone();
1084
1085         // clone arrays. Can't do with utility because of bug in Cloneable
1086         other.formats = formats.clone(); // shallow clone
1087         for (int i = 0; i < formats.length; ++i) {
1088             if (formats[i] != null)
1089                 other.formats[i] = (Format)formats[i].clone();
1090         }
1091         // for primitives or immutables, shallow clone is enough
1092         other.offsets = offsets.clone();
1093         other.argumentNumbers = argumentNumbers.clone();
1094
1095         return other;
1096     }
1097
1098     /**
1099     * Equality comparison between two message format objects
1100     */
1101     public boolean equals(Object obj) {
1102         if (this == obj)                // quick check
1103             return true;
1104         if (obj == null || getClass() != obj.getClass())
1105             return false;
1106         MessageFormat other = (MessageFormat) obj;
1107         return (maxOffset == other.maxOffset
1108             && pattern.equals(other.pattern)
1109             && ((locale != null && locale.equals(other.locale))
1110                 || (locale == null && other.locale == null))
1111             && Arrays.equals(offsets, other.offsets)
1112             && Arrays.equals(argumentNumbers, other.argumentNumbers)
1113             && Arrays.equals(formats, other.formats));
1114     }
1115
1116     /**

```

```

1117     * Generates a hash code for the message format object.
1118     */
1119     public int hashCode() {
1120         return pattern.hashCode(); // enough for reasonable distribution
1121     }
1122
1123
1124     /**
1125     * Defines constants that are used as attribute keys in the
1126     * <code>AttributedCharacterIterator</code> returned
1127     * from <code>MessageFormat.formatToCharacterIterator</code>.
1128     *
1129     * @since 1.4
1130     */
1131     public static class Field extends Format.Field {
1132
1133         // Proclaim serial compatibility with 1.4 FCS
1134         private static final long serialVersionUID = 7899943957617360810L;
1135
1136         /**
1137         * Creates a Field with the specified name.
1138         *
1139         * @param name Name of the attribute
1140         */
1141         protected Field(String name) {
1142             super(name);
1143         }
1144
1145         /**
1146         * Resolves instances being deserialized to the predefined constants.
1147         *
1148         * @throws InvalidObjectException if the constant could not be
1149         *         resolved.
1150         * @return resolved MessageFormat.Field constant
1151         */
1152         protected Object readResolve() throws InvalidObjectException {
1153             if (this.getClass() != MessageFormat.Field.class) {
1154                 throw new InvalidObjectException("subclass didn't correctly implement readResolve")
1155                 ;
1156             }
1157
1158             return ARGUMENT;
1159         }
1160
1161         //
1162         // The constants
1163         //
1164
1165         /**
1166         * Constant identifying a portion of a message that was generated
1167         * from an argument passed into <code>formatToCharacterIterator</code>.
1168         * The value associated with the key will be an <code>Integer</code>
1169         * indicating the index in the <code>arguments</code> array of the

```

```

1169     * argument from which the text was generated.
1170     */
1171     public final static Field ARGUMENT =
1172         new Field("message argument field");
1173 }
1174
1175 // =====privates=====
1176
1177 /**
1178  * The locale to use for formatting numbers and dates.
1179  * @serial
1180  */
1181 private Locale locale;
1182
1183 /**
1184  * The string that the formatted values are to be plugged into. In other words, this
1185  * is the pattern supplied on construction with all of the {} expressions taken out.
1186  * @serial
1187  */
1188 private String pattern = "";
1189
1190 /** The initially expected number of subformats in the format */
1191 private static final int INITIAL_FORMATS = 10;
1192
1193 /**
1194  * An array of formatters, which are used to format the arguments.
1195  * @serial
1196  */
1197 private Format[] formats = new Format[INITIAL_FORMATS];
1198
1199 /**
1200  * The positions where the results of formatting each argument are to be inserted
1201  * into the pattern.
1202  * @serial
1203  */
1204 private int[] offsets = new int[INITIAL_FORMATS];
1205
1206 /**
1207  * The argument numbers corresponding to each formatter. (The formatters are stored
1208  * in the order they occur in the pattern, not in the order in which the arguments
1209  * are specified.)
1210  * @serial
1211  */
1212 private int[] argumentNumbers = new int[INITIAL_FORMATS];
1213
1214 /**
1215  * One less than the number of entries in <code>offsets</code>. Can also be thought of
1216  * as the index of the highest-numbered element in <code>offsets</code> that is being used.
1217  * All of these arrays should have the same number of elements being used as <code>offsets</code>
1218  * does, and so this variable suffices to tell us how many entries are in all of them.
1219  * @serial
1220  */

```

```

1221     private int maxOffset = -1;
1222
1223     /**
1224      * Internal routine used by format. If <code>characterIterators</code> is
1225      * non-null, AttributedCharacterIterator will be created from the
1226      * subformats as necessary. If <code>characterIterators</code> is null
1227      * and <code>fp</code> is non-null and identifies
1228      * <code>Field.MESSAGE_ARGUMENT</code>, the location of
1229      * the first replaced argument will be set in it.
1230      *
1231      * @exception IllegalArgumentException if an argument in the
1232      *         <code>arguments</code> array is not of the type
1233      *         expected by the format element(s) that use it.
1234      */
1235     private StringBuffer subformat(Object[] arguments, StringBuffer result,
1236                                   FieldPosition fp, List<AttributedCharacterIterator>
1237                                   characterIterators) {
1238         // note: this implementation assumes a fast substring & index.
1239         // if this is not true, would be better to append chars one by one.
1240         int lastOffset = 0;
1241         int last = result.length();
1242         for (int i = 0; i <= maxOffset; ++i) {
1243             result.append(pattern.substring(lastOffset, offsets[i]));
1244             lastOffset = offsets[i];
1245             int argumentNumber = argumentNumbers[i];
1246             if (arguments == null || argumentNumber >= arguments.length) {
1247                 result.append('{').append(argumentNumber).append('}');
1248                 continue;
1249             }
1250             // int argRecursion = ((recursionProtection >> (argumentNumber*2)) & 0x3);
1251             if (false) { // if (argRecursion == 3){
1252                 // prevent loop!!!
1253                 result.append('\uFFFD');
1254             } else {
1255                 Object obj = arguments[argumentNumber];
1256                 String arg = null;
1257                 Format subFormatter = null;
1258                 if (obj == null) {
1259                     arg = "null";
1260                 } else if (formats[i] != null) {
1261                     subFormatter = formats[i];
1262                     if (subFormatter instanceof ChoiceFormat) {
1263                         arg = formats[i].format(obj);
1264                         if (arg.indexOf('{') >= 0) {
1265                             subFormatter = new MessageFormat(arg, locale);
1266                             obj = arguments;
1267                             arg = null;
1268                         }
1269                     }
1270                 } else if (obj instanceof Number) {
1271                     // format number if can
1272                     subFormatter = NumberFormat.getInstance(locale);
1273                 } else if (obj instanceof Date) {

```

```
1273         // format a Date if can
1274         subFormatter = DateFormat.getDateTimeInstance(
1275             DateFormat.SHORT, DateFormat.SHORT, locale); //fix
1276     } else if (obj instanceof String) {
1277         arg = (String) obj;
1278
1279     } else {
1280         arg = obj.toString();
1281         if (arg == null) arg = "null";
1282     }
1283
1284     // At this point we are in two states, either subFormatter
1285     // is non-null indicating we should format obj using it,
1286     // or arg is non-null and we should use it as the value.
1287
1288     if (characterIterators != null) {
1289         // If characterIterators is non-null, it indicates we need
1290         // to get the CharacterIterator from the child formatter.
1291         if (last != result.length()) {
1292             characterIterators.add(
1293                 createAttributedCharacterIterator(result.substring(
1294                     last)));
1295
1296             last = result.length();
1297         }
1298         if (subFormatter != null) {
1299             AttributedCharacterIterator subIterator =
1300                 subFormatter.formatToCharacterIterator(obj);
1301
1302             append(result, subIterator);
1303             if (last != result.length()) {
1304                 characterIterators.add(
1305                     createAttributedCharacterIterator(
1306                         subIterator, Field.ARGUMENT,
1307                         Integer.valueOf(argumentNumber)));
1308                 last = result.length();
1309             }
1310             arg = null;
1311         }
1312         if (arg != null && arg.length() > 0) {
1313             result.append(arg);
1314             characterIterators.add(
1315                 createAttributedCharacterIterator(
1316                     arg, Field.ARGUMENT,
1317                     Integer.valueOf(argumentNumber)));
1318             last = result.length();
1319         }
1320     } else {
1321         if (subFormatter != null) {
1322             arg = subFormatter.format(obj);
1323         }
1324         last = result.length();
1325         result.append(arg);
1326     }
```



```

1326         if (i == 0 && fp != null && Field.ARGUMENT.equals(
1327             fp.getFieldAttribute())) {
1328             fp.setBeginIndex(last);
1329             fp.setEndIndex(result.length());
1330         }
1331         last = result.length();
1332     }
1333 }
1334 }
1335 result.append(pattern.substring(lastOffset, pattern.length()));
1336 if (characterIterators != null && last != result.length()) {
1337     characterIterators.add(createAttributedCharacterIterator(
1338         result.substring(last)));
1339 }
1340 return result;
1341 }
1342
1343 /**
1344  * Convenience method to append all the characters in
1345  * <code>iterator</code> to the StringBuffer <code>result</code>.
1346  */
1347 private void append(StringBuffer result, CharacterIterator iterator) {
1348     if (iterator.first() != CharacterIterator.DONE) {
1349         char aChar;
1350
1351         result.append(iterator.first());
1352         while ((aChar = iterator.next()) != CharacterIterator.DONE) {
1353             result.append(aChar);
1354         }
1355     }
1356 }
1357
1358 // Indices for segments
1359 private static final int SEG_RAW      = 0;
1360 private static final int SEG_INDEX    = 1;
1361 private static final int SEG_TYPE     = 2;
1362 private static final int SEG_MODIFIER = 3; // modifier or subformat
1363
1364 // Indices for type keywords
1365 private static final int TYPE_NULL    = 0;
1366 private static final int TYPE_NUMBER = 1;
1367 private static final int TYPE_DATE    = 2;
1368 private static final int TYPE_TIME    = 3;
1369 private static final int TYPE_CHOICE  = 4;
1370
1371 private static final String[] TYPE_KEYWORDS = {
1372     "",
1373     "number",
1374     "date",
1375     "time",
1376     "choice"
1377 };
1378

```

```

1379 // Indices for number modifiers
1380 private static final int MODIFIER_DEFAULT = 0; // common in number and date-time
1381 private static final int MODIFIER_CURRENCY = 1;
1382 private static final int MODIFIER_PERCENT = 2;
1383 private static final int MODIFIER_INTEGER = 3;
1384
1385 private static final String[] NUMBER_MODIFIER_KEYWORDS = {
1386     "",
1387     "currency",
1388     "percent",
1389     "integer"
1390 };
1391
1392 // Indices for date-time modifiers
1393 private static final int MODIFIER_SHORT = 1;
1394 private static final int MODIFIER_MEDIUM = 2;
1395 private static final int MODIFIER_LONG = 3;
1396 private static final int MODIFIER_FULL = 4;
1397
1398 private static final String[] DATE_TIME_MODIFIER_KEYWORDS = {
1399     "",
1400     "short",
1401     "medium",
1402     "long",
1403     "full"
1404 };
1405
1406 // Date-time style values corresponding to the date-time modifiers.
1407 private static final int[] DATE_TIME_MODIFIERS = {
1408     DateFormat.DEFAULT,
1409     DateFormat.SHORT,
1410     DateFormat.MEDIUM,
1411     DateFormat.LONG,
1412     DateFormat.FULL,
1413 };
1414
1415 private void makeFormat(int position, int offsetNumber,
1416     StringBuilder[] textSegments)
1417 {
1418     String[] segments = new String[textSegments.length];
1419     for (int i = 0; i < textSegments.length; i++) {
1420         StringBuilder oneseg = textSegments[i];
1421         segments[i] = (oneseg != null) ? oneseg.toString() : "";
1422     }
1423
1424     // get the argument number
1425     int argumentNumber;
1426     try {
1427         argumentNumber = Integer.parseInt(segments[SEG_INDEX]); // always unlocalized!
1428     } catch (NumberFormatException e) {
1429         throw new IllegalArgumentException("can't parse argument number: "
1430             + segments[SEG_INDEX], e);
1431     }

```



```

1485         throw e;
1486     }
1487     break;
1488 }
1489 break;
1490
1491 case TYPE_DATE:
1492 case TYPE_TIME:
1493     int mod = findKeyword(segments[SEG_MODIFIER], DATE_TIME_MODIFIER_KEYWORDS);
1494     if (mod >= 0 && mod < DATE_TIME_MODIFIER_KEYWORDS.length) {
1495         if (type == TYPE_DATE) {
1496             newFormat = DateFormat.getDateInstance(DATE_TIME_MODIFIERS[mod],
1497                                                         locale);
1498         } else {
1499             newFormat = DateFormat.getTimeInstance(DATE_TIME_MODIFIERS[mod],
1500                                                         locale);
1501         }
1502     } else {
1503         // SimpleDateFormat pattern
1504         try {
1505             newFormat = new SimpleDateFormat(segments[SEG_MODIFIER], locale);
1506         } catch (IllegalArgumentException e) {
1507             maxOffset = oldMaxOffset;
1508             throw e;
1509         }
1510     }
1511     break;
1512
1513 case TYPE_CHOICE:
1514     try {
1515         // ChoiceFormat pattern
1516         newFormat = new ChoiceFormat(segments[SEG_MODIFIER]);
1517     } catch (Exception e) {
1518         maxOffset = oldMaxOffset;
1519         throw new IllegalArgumentException("Choice Pattern incorrect: "
1520                                           + segments[SEG_MODIFIER], e);
1521     }
1522     break;
1523
1524 default:
1525     maxOffset = oldMaxOffset;
1526     throw new IllegalArgumentException("unknown format type: " +
1527                                       segments[SEG_TYPE]);
1528 }
1529 }
1530 formats[offsetNumber] = newFormat;
1531 }
1532
1533 private static final int findKeyword(String s, String[] list) {
1534     for (int i = 0; i < list.length; ++i) {
1535         if (s.equals(list[i]))
1536             return i;
1537     }

```

```

1538
1539     // Try trimmed lowercase.
1540     String ls = s.trim().toLowerCase(Locale.ROOT);
1541     if (ls != s) {
1542         for (int i = 0; i < list.length; ++i) {
1543             if (ls.equals(list[i]))
1544                 return i;
1545         }
1546     }
1547     return -1;
1548 }
1549
1550 private static final void copyAndFixQuotes(String source, int start, int end,
1551                                           StringBuilder target) {
1552     boolean quoted = false;
1553
1554     for (int i = start; i < end; ++i) {
1555         char ch = source.charAt(i);
1556         if (ch == '{') {
1557             if (!quoted) {
1558                 target.append('\\');
1559                 quoted = true;
1560             }
1561             target.append(ch);
1562         } else if (ch == '\\') {
1563             target.append("\\");
1564         } else {
1565             if (quoted) {
1566                 target.append('\\');
1567                 quoted = false;
1568             }
1569             target.append(ch);
1570         }
1571     }
1572     if (quoted) {
1573         target.append('\\');
1574     }
1575 }
1576
1577 /**
1578  * After reading an object from the input stream, do a simple verification
1579  * to maintain class invariants.
1580  * @throws InvalidObjectException if the objects read from the stream is invalid.
1581  */
1582 private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
1583     in.defaultReadObject();
1584     boolean isValid = maxOffset >= -1
1585         && formats.length > maxOffset
1586         && offsets.length > maxOffset
1587         && argumentNumbers.length > maxOffset;
1588     if (isValid) {
1589         int lastOffset = pattern.length() + 1;
1590         for (int i = maxOffset; i >= 0; --i) {

```

```

1591         if ((offsets[i] < 0) || (offsets[i] > lastOffset)) {
1592             isValid = false;
1593             break;
1594         } else {
1595             lastOffset = offsets[i];
1596         }
1597     }
1598 }
1599 if (!isValid) {
1600     throw new InvalidObjectException("Could not reconstruct MessageFormat from corrupt
1601                                     stream.");
1602 }
1603 }

```

2.24 Normalizer.java

Secuencia 33: java/text/Normalizer.java

```

1  /*
2  * Copyright (c) 2005, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *****
28 * (C) Copyright IBM Corp. 1996-2005 - All Rights Reserved *
29 * *
30 * The original version of this source code and documentation is copyrighted *
31 * and owned by IBM, These materials are provided under terms of a License *
32 * Agreement between IBM and Sun. This technology is protected by multiple *
33 * US and International patents. This notice and attribution to IBM may not *
34 * to removed. *
35 *****

```

```

36  */
37
38  package java.text;
39
40  import sun.text.normalizer.NormalizerBase;
41  import sun.text.normalizer.NormalizerImpl;
42
43  /**
44   * This class provides the method normalize which transforms Unicode
45   * text into an equivalent composed or decomposed form, allowing for easier
46   * sorting and searching of text.
47   * The normalize method supports the standard normalization forms
48   * described in
49   * 
49   \* <a href="http://www.unicode.org/unicode/reports/tr15/tr15-23.html">
50   * Unicode Standard Annex #15 &mdash; Unicode Normalization Forms.
51   * <p>
52   * Characters with accents or other adornments can be encoded in
53   * several different ways in Unicode. For example, take the character A-acute.
54   * In Unicode, this can be encoded as a single character (the "composed" form):
55   *
56   * <pre>
57   *      U+00C1    LATIN CAPITAL LETTER A WITH ACUTE</pre>
58   *
59   * or as two separate characters (the "decomposed" form):
60   *
61   * <pre>
62   *      U+0041    LATIN CAPITAL LETTER A
63   *      U+0301    COMBINING ACUTE ACCENT</pre>
64   *
65   * To a user of your program, however, both of these sequences should be
66   * treated as the same "user-level" character "A with acute accent". When you
67   * are searching or comparing text, you must ensure that these two sequences are
68   * treated as equivalent. In addition, you must handle characters with more than
69   * one accent. Sometimes the order of a character's combining accents is
70   * significant, while in other cases accent sequences in different orders are
71   * really equivalent.
72   * <p>
73   * Similarly, the string "ffi" can be encoded as three separate letters:
74   *
75   * <pre>
76   *      U+0066    LATIN SMALL LETTER F
77   *      U+0066    LATIN SMALL LETTER F
78   *      U+0069    LATIN SMALL LETTER I</pre>
79   *
80   * or as the single character
81   *
82   * <pre>
83   *      U+FB03    LATIN SMALL LIGATURE FFI</pre>
84   *
85   * The ffi ligature is not a distinct semantic character, and strictly speaking
86   * it shouldn't be in Unicode at all, but it was included for compatibility
87   * with existing character sets that already provided it. The Unicode standard
88   * identifies such characters by giving them "compatibility" decompositions

```

```

89  * into the corresponding semantic characters. When sorting and searching, you
90  * will often want to use these mappings.
91  * <p>
92  * The <code>normalize</code> method helps solve these problems by transforming
93  * text into the canonical composed and decomposed forms as shown in the first
94  * example above. In addition, you can have it perform compatibility
95  * decompositions so that you can treat compatibility characters the same as
96  * their equivalents.
97  * Finally, the <code>normalize</code> method rearranges accents into the
98  * proper canonical order, so that you do not have to worry about accent
99  * rearrangement on your own.
100 * <p>
101 * The W3C generally recommends to exchange texts in NFC.
102 * Note also that most legacy character encodings use only precomposed forms and
103 * often do not encode any combining marks by themselves. For conversion to such
104 * character encodings the Unicode text needs to be normalized to NFC.
105 * For more usage examples, see the Unicode Standard Annex.
106 *
107 * @since 1.6
108 */
109 public final class Normalizer {
110
111     private Normalizer() {};
112
113     /**
114      * This enum provides constants of the four Unicode normalization forms
115      * that are described in
116      * <a href="http://www.unicode.org/unicode/reports/tr15/tr15-23.html">
117      * Unicode Standard Annex #15 &mdash; Unicode Normalization Forms</a>
118      * and two methods to access them.
119      *
120      * @since 1.6
121      */
122     public static enum Form {
123
124         /**
125          * Canonical decomposition.
126          */
127         NFD,
128
129         /**
130          * Canonical decomposition, followed by canonical composition.
131          */
132         NFC,
133
134         /**
135          * Compatibility decomposition.
136          */
137         NFKD,
138
139         /**
140          * Compatibility decomposition, followed by canonical composition.
141          */

```



```

142     NFKC
143 }
144
145 /**
146  * Normalize a sequence of char values.
147  * The sequence will be normalized according to the specified normalization
148  * from.
149  * @param src      The sequence of char values to normalize.
150  * @param form      The normalization form; one of
151  *                  {@link java.text.Normalizer.Form#NFC},
152  *                  {@link java.text.Normalizer.Form#NFD},
153  *                  {@link java.text.Normalizer.Form#NFKC},
154  *                  {@link java.text.Normalizer.Form#NFKD}
155  * @return The normalized String
156  * @throws NullPointerException If <code>src</code> or <code>form</code>
157  * is null.
158  */
159 public static String normalize(CharSequence src, Form form) {
160     return NormalizerBase.normalize(src.toString(), form);
161 }
162
163 /**
164  * Determines if the given sequence of char values is normalized.
165  * @param src      The sequence of char values to be checked.
166  * @param form      The normalization form; one of
167  *                  {@link java.text.Normalizer.Form#NFC},
168  *                  {@link java.text.Normalizer.Form#NFD},
169  *                  {@link java.text.Normalizer.Form#NFKC},
170  *                  {@link java.text.Normalizer.Form#NFKD}
171  * @return true if the sequence of char values is normalized;
172  * false otherwise.
173  * @throws NullPointerException If <code>src</code> or <code>form</code>
174  * is null.
175  */
176 public static boolean isNormalized(CharSequence src, Form form) {
177     return NormalizerBase.isNormalized(src.toString(), form);
178 }
179 }

```

2.25 NumberFormat.java

Secuencia 34: java/text/NumberFormat.java

```

1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *

```

```
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27   * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28   * (C) Copyright IBM Corp. 1996 - 1998 - All Rights Reserved
29   *
30   * The original version of this source code and documentation is copyrighted
31   * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32   * materials are provided under terms of a License Agreement between Taligent
33   * and Sun. This technology is protected by multiple US and International
34   * patents. This notice and attribution to Taligent may not be removed.
35   * Taligent is a registered trademark of Taligent, Inc.
36   *
37   */
38
39  package java.text;
40
41  import java.io.InvalidObjectException;
42  import java.io.IOException;
43  import java.io.ObjectInputStream;
44  import java.io.ObjectOutputStream;
45  import java.math.BigInteger;
46  import java.math.RoundingMode;
47  import java.text.spi.NumberFormatProvider;
48  import java.util.Currency;
49  import java.util.HashMap;
50  import java.util.Hashtable;
51  import java.util.Locale;
52  import java.util.Map;
53  import java.util.ResourceBundle;
54  import java.util.concurrent.atomic.AtomicInteger;
55  import java.util.concurrent.atomic.AtomicLong;
56  import java.util.spi.LocaleServiceProvider;
57  import sun.util.locale.provider.LocaleProviderAdapter;
58  import sun.util.locale.provider.LocaleServiceProviderPool;
59
60  /**
61   * <code>NumberFormat</code> is the abstract base class for all number
62   * formats. This class provides the interface for formatting and parsing
63   * numbers. <code>NumberFormat</code> also provides methods for determining
64   * which locales have number formats, and what their names are.
```

```
65 *
66 * <p>
67 * <code>NumberFormat</code> helps you to format and parse numbers for any locale.
68 * Your code can be completely independent of the locale conventions for
69 * decimal points, thousands-separators, or even the particular decimal
70 * digits used, or whether the number format is even decimal.
71 *
72 * <p>
73 * To format a number for the current Locale, use one of the factory
74 * class methods:
75 * <blockquote>
76 * <pre>{@code
77 * myString = NumberFormat.getInstance().format(myNumber);
78 * }</pre>
79 * </blockquote>
80 * If you are formatting multiple numbers, it is
81 * more efficient to get the format and use it multiple times so that
82 * the system doesn't have to fetch the information about the local
83 * language and country conventions multiple times.
84 * <blockquote>
85 * <pre>{@code
86 * NumberFormat nf = NumberFormat.getInstance();
87 * for (int i = 0; i < myNumber.length; ++i) {
88 *     output.println(nf.format(myNumber[i]) + " ");
89 * }
90 * }</pre>
91 * </blockquote>
92 * To format a number for a different Locale, specify it in the
93 * call to <code>getInstance</code>.
94 * <blockquote>
95 * <pre>{@code
96 * NumberFormat nf = NumberFormat.getInstance(Locale.FRENCH);
97 * }</pre>
98 * </blockquote>
99 * You can also use a <code>NumberFormat</code> to parse numbers:
100 * <blockquote>
101 * <pre>{@code
102 * myNumber = nf.parse(myString);
103 * }</pre>
104 * </blockquote>
105 * Use <code>getInstance</code> or <code>getNumberInstance</code> to get the
106 * normal number format. Use <code>getIntegerInstance</code> to get an
107 * integer number format. Use <code>getCurrencyInstance</code> to get the
108 * currency number format. And use <code>getPercentInstance</code> to get a
109 * format for displaying percentages. With this format, a fraction like
110 * 0.53 is displayed as 53%.
111 *
112 * <p>
113 * You can also control the display of numbers with such methods as
114 * <code>setMinimumFractionDigits</code>.
115 * If you want even more control over the format or parsing,
116 * or want to give your users more control,
117 * you can try casting the <code>NumberFormat</code> you get from the factory methods
```

```

118 * to a DecimalFormat. This will work for the vast majority
119 * of locales; just remember to put it in a try block in case you
120 * encounter an unusual one.
121 *
122 * <p>
123 * NumberFormat and DecimalFormat are designed such that some controls
124 * work for formatting and others work for parsing. The following is
125 * the detailed description for each these control methods,
126 * <p>
127 * setParseIntegerOnly : only affects parsing, e.g.
128 * if true, "3456.78" &rarr; 3456 (and leaves the parse position just after index 6)
129 * if false, "3456.78" &rarr; 3456.78 (and leaves the parse position just after index 8)
130 * This is independent of formatting. If you want to not show a decimal point
131 * where there might be no digits after the decimal point, use
132 * setDecimalSeparatorAlwaysShown.
133 * <p>
134 * setDecimalSeparatorAlwaysShown : only affects formatting, and only where
135 * there might be no digits after the decimal point, such as with a pattern
136 * like "#,##0.##", e.g.,
137 * if true, 3456.00 &rarr; "3,456."
138 * if false, 3456.00 &rarr; "3456"
139 * This is independent of parsing. If you want parsing to stop at the decimal
140 * point, use setParseIntegerOnly.
141 *
142 * <p>
143 * You can also use forms of the parse and format
144 * methods with ParsePosition and FieldPosition to
145 * allow you to:
146 * <ul>
147 * <li> progressively parse through pieces of a string
148 * <li> align the decimal point and other areas
149 * </ul>
150 * For example, you can align numbers in two ways:
151 * <ol>
152 * <li> If you are using a monospaced font with spacing for alignment,
153 *     you can pass the FieldPosition in your format call, with
154 *     field = INTEGER_FIELD. On output,
155 *     getEndIndex will be set to the offset between the
156 *     last character of the integer and the decimal. Add
157 *     (desiredSpaceCount - getEndIndex) spaces at the front of the string.
158 *
159 * <li> If you are using proportional fonts,
160 *     instead of padding with spaces, measure the width
161 *     of the string in pixels from the start to getEndIndex.
162 *     Then move the pen by
163 *     (desiredPixelWidth - widthToAlignmentPoint) before drawing the text.
164 *     It also works where there is no decimal, but possibly additional
165 *     characters at the end, e.g., with parentheses in negative
166 *     numbers: "(12)" for -12.
167 * </ol>
168 *
169 * <h3><a name="synchronization">Synchronization</a></h3>
170 *

```

```

171 * <p>
172 * Number formats are generally not synchronized.
173 * It is recommended to create separate format instances for each thread.
174 * If multiple threads access a format concurrently, it must be synchronized
175 * externally.
176 *
177 * @see      DecimalFormat
178 * @see      ChoiceFormat
179 * @author    Mark Davis
180 * @author    Helena Shih
181 */
182 public abstract class NumberFormat extends Format {
183
184     /**
185      * Field constant used to construct a FieldPosition object. Signifies that
186      * the position of the integer part of a formatted number should be returned.
187      * @see java.text.FieldPosition
188      */
189     public static final int INTEGER_FIELD = 0;
190
191     /**
192      * Field constant used to construct a FieldPosition object. Signifies that
193      * the position of the fraction part of a formatted number should be returned.
194      * @see java.text.FieldPosition
195      */
196     public static final int FRACTION_FIELD = 1;
197
198     /**
199      * Sole constructor. (For invocation by subclass constructors, typically
200      * implicit.)
201      */
202     protected NumberFormat() {
203     }
204
205     /**
206      * Formats a number and appends the resulting text to the given string
207      * buffer.
208      * The number can be of any subclass of {@link java.lang.Number}.
209      * <p>
210      * This implementation extracts the number's value using
211      * {@link java.lang.Number#longValue()} for all integral type values that
212      * can be converted to long without loss of information,
213      * including BigInteger values with a
214      * {@link java.math.BigInteger#bitLength() bit length} of less than 64,
215      * and {@link java.lang.Number#doubleValue()} for all other types. It
216      * then calls
217      * {@link #format(long,java.lang.StringBuffer,java.text.FieldPosition)}
218      * or {@link #format(double,java.lang.StringBuffer,java.text.FieldPosition)}.
219      * This may result in loss of magnitude information and precision for
220      * BigInteger and BigDecimal values.
221      * @param number    the number to format
222      * @param toAppendTo the StringBuffer to which the formatted
223      *                  text is to be appended

```

```

224 * @param pos      On input: an alignment field, if desired.
225 *                On output: the offsets of the alignment field.
226 * @return         the value passed in as <code>toAppendTo</code>
227 * @exception      IllegalArgumentException if <code>number</code> is
228 *                null or not an instance of <code>Number</code>.
229 * @exception      NullPointerException if <code>toAppendTo</code> or
230 *                <code>pos</code> is null
231 * @exception      ArithmeticException if rounding is needed with rounding
232 *                mode being set to RoundingMode.UNNECESSARY
233 * @see            java.text.FieldPosition
234 */
235 @Override
236 public StringBuffer format(Object number,
237                             StringBuffer toAppendTo,
238                             FieldPosition pos) {
239     if (number instanceof Long || number instanceof Integer ||
240         number instanceof Short || number instanceof Byte ||
241         number instanceof AtomicInteger || number instanceof AtomicLong ||
242         (number instanceof BigInteger &&
243          ((BigInteger)number).bitLength() < 64)) {
244         return format(((Number)number).longValue(), toAppendTo, pos);
245     } else if (number instanceof Number) {
246         return format(((Number)number).doubleValue(), toAppendTo, pos);
247     } else {
248         throw new IllegalArgumentException("Cannot format given Object as a Number");
249     }
250 }
251
252 /**
253  * Parses text from a string to produce a <code>Number</code>.
254  * <p>
255  * The method attempts to parse text starting at the index given by
256  * <code>pos</code>.
257  * If parsing succeeds, then the index of <code>pos</code> is updated
258  * to the index after the last character used (parsing does not necessarily
259  * use all characters up to the end of the string), and the parsed
260  * number is returned. The updated <code>pos</code> can be used to
261  * indicate the starting point for the next call to this method.
262  * If an error occurs, then the index of <code>pos</code> is not
263  * changed, the error index of <code>pos</code> is set to the index of
264  * the character where the error occurred, and null is returned.
265  * <p>
266  * See the {@link #parse(String, ParsePosition)} method for more information
267  * on number parsing.
268  *
269  * @param source A <code>String</code>, part of which should be parsed.
270  * @param pos A <code>ParsePosition</code> object with index and error
271  *            index information as described above.
272  * @return A <code>Number</code> parsed from the string. In case of
273  *         error, returns null.
274  * @exception NullPointerException if <code>pos</code> is null.
275  */
276 @Override

```

```

277     public final Object parseObject(String source, ParsePosition pos) {
278         return parse(source, pos);
279     }
280
281     /**
282      * Specialization of format.
283      *
284      * @param number the double number to format
285      * @return the formatted String
286      * @exception      ArithmeticException if rounding is needed with rounding
287      *                  mode being set to RoundingMode.UNNECESSARY
288      * @see java.text.Format#format
289      */
290     public final String format(double number) {
291         // Use fast-path for double result if that works
292         String result = fastFormat(number);
293         if (result != null)
294             return result;
295
296         return format(number, new StringBuffer(),
297             DontCareFieldPosition.INSTANCE).toString();
298     }
299
300     /**
301      * fastFormat() is supposed to be implemented in concrete subclasses only.
302      * Default implem always returns null.
303      */
304     String fastFormat(double number) { return null; }
305
306     /**
307      * Specialization of format.
308      *
309      * @param number the long number to format
310      * @return the formatted String
311      * @exception      ArithmeticException if rounding is needed with rounding
312      *                  mode being set to RoundingMode.UNNECESSARY
313      * @see java.text.Format#format
314      */
315     public final String format(long number) {
316         return format(number, new StringBuffer(),
317             DontCareFieldPosition.INSTANCE).toString();
318     }
319
320     /**
321      * Specialization of format.
322      *
323      * @param number    the double number to format
324      * @param toAppendTo the StringBuffer to which the formatted text is to be
325      *                  appended
326      * @param pos        the field position
327      * @return the formatted StringBuffer
328      * @exception      ArithmeticException if rounding is needed with rounding
329      *                  mode being set to RoundingMode.UNNECESSARY

```

```

330     * @see java.text.Format#format
331     */
332     public abstract StringBuffer format(double number,
333                                     StringBuffer toAppendTo,
334                                     FieldPosition pos);
335
336     /**
337     * Specialization of format.
338     *
339     * @param number    the long number to format
340     * @param toAppendTo the StringBuffer to which the formatted text is to be
341     *                  appended
342     * @param pos        the field position
343     * @return the formatted StringBuffer
344     * @exception ArithmeticException if rounding is needed with rounding
345     *                  mode being set to RoundingMode.UNNECESSARY
346     * @see java.text.Format#format
347     */
348     public abstract StringBuffer format(long number,
349                                     StringBuffer toAppendTo,
350                                     FieldPosition pos);
351
352     /**
353     * Returns a Long if possible (e.g., within the range [Long.MIN_VALUE,
354     * Long.MAX_VALUE] and with no decimals), otherwise a Double.
355     * If IntegerOnly is set, will stop at a decimal
356     * point (or equivalent; e.g., for rational numbers "1 2/3", will stop
357     * after the 1).
358     * Does not throw an exception; if no object can be parsed, index is
359     * unchanged!
360     *
361     * @param source the String to parse
362     * @param parsePosition the parse position
363     * @return the parsed value
364     * @see java.text.NumberFormat#isParseIntegerOnly
365     * @see java.text.Format#parseObject
366     */
367     public abstract Number parse(String source, ParsePosition parsePosition);
368
369     /**
370     * Parses text from the beginning of the given string to produce a number.
371     * The method may not use the entire text of the given string.
372     * <p>
373     * See the {@link #parse(String, ParsePosition)} method for more information
374     * on number parsing.
375     *
376     * @param source A <code>String</code> whose beginning should be parsed.
377     * @return A <code>Number</code> parsed from the string.
378     * @exception ParseException if the beginning of the specified string
379     *                  cannot be parsed.
380     */
381     public Number parse(String source) throws ParseException {
382         ParsePosition parsePosition = new ParsePosition(0);

```



```

383     Number result = parse(source, parsePosition);
384     if (parsePosition.index == 0) {
385         throw new ParseException("Unparseable number: \"" + source + "\"",
386                                 parsePosition.errorIndex);
387     }
388     return result;
389 }
390
391 /**
392  * Returns true if this format will parse numbers as integers only.
393  * For example in the English locale, with ParseIntegerOnly true, the
394  * string "1234." would be parsed as the integer value 1234 and parsing
395  * would stop at the "." character. Of course, the exact format accepted
396  * by the parse operation is locale dependant and determined by sub-classes
397  * of NumberFormat.
398  *
399  * @return {@code true} if numbers should be parsed as integers only;
400  *         {@code false} otherwise
401  */
402 public boolean isParseIntegerOnly() {
403     return parseIntegerOnly;
404 }
405
406 /**
407  * Sets whether or not numbers should be parsed as integers only.
408  *
409  * @param value {@code true} if numbers should be parsed as integers only;
410  *              {@code false} otherwise
411  * @see #isParseIntegerOnly
412  */
413 public void setParseIntegerOnly(boolean value) {
414     parseIntegerOnly = value;
415 }
416
417 //===== Locale Stuff =====
418
419 /**
420  * Returns a general-purpose number format for the current default
421  * {@link java.util.Locale.Category#FORMAT FORMAT} locale.
422  * This is the same as calling
423  * {@link #getNumberInstance() getNumberInstance()}.
424  *
425  * @return the {@code NumberFormat} instance for general-purpose number
426  *         formatting
427  */
428 public final static NumberFormat getInstance() {
429     return getInstance(Locale.getDefault(Locale.Category.FORMAT), NUMBERSTYLE);
430 }
431
432 /**
433  * Returns a general-purpose number format for the specified locale.
434  * This is the same as calling
435  * {@link #getNumberInstance(java.util.Locale) getNumberInstance(inLocale)}.

```

```

436     *
437     * @param inLocale the desired locale
438     * @return the {@code NumberFormat} instance for general-purpose number
439     * formatting
440     */
441     public static NumberFormat getInstance(Locale inLocale) {
442         return getInstance(inLocale, NUMBERSTYLE);
443     }
444
445     /**
446     * Returns a general-purpose number format for the current default
447     * {@link java.util.Locale.Category#FORMAT FORMAT} locale.
448     * <p>This is equivalent to calling
449     * {@link #getNumberInstance(Locale)
450     *     getNumberInstance(Locale.getDefault(Locale.Category.FORMAT))}.
451     *
452     * @return the {@code NumberFormat} instance for general-purpose number
453     * formatting
454     * @see java.util.Locale#getDefault(java.util.Locale.Category)
455     * @see java.util.Locale.Category#FORMAT
456     */
457     public final static NumberFormat getNumberInstance() {
458         return getInstance(Locale.getDefault(Locale.Category.FORMAT), NUMBERSTYLE);
459     }
460
461     /**
462     * Returns a general-purpose number format for the specified locale.
463     *
464     * @param inLocale the desired locale
465     * @return the {@code NumberFormat} instance for general-purpose number
466     * formatting
467     */
468     public static NumberFormat getNumberInstance(Locale inLocale) {
469         return getInstance(inLocale, NUMBERSTYLE);
470     }
471
472     /**
473     * Returns an integer number format for the current default
474     * {@link java.util.Locale.Category#FORMAT FORMAT} locale. The
475     * returned number format is configured to round floating point numbers
476     * to the nearest integer using half-even rounding (see {@link
477     * java.math.RoundingMode#HALF_EVEN RoundingMode.HALF_EVEN}) for formatting,
478     * and to parse only the integer part of an input string (see {@link
479     * #isParseIntegerOnly isParseIntegerOnly}).
480     * <p>This is equivalent to calling
481     * {@link #getIntegerInstance(Locale)
482     *     getIntegerInstance(Locale.getDefault(Locale.Category.FORMAT))}.
483     *
484     * @see #getRoundingMode()
485     * @see java.util.Locale#getDefault(java.util.Locale.Category)
486     * @see java.util.Locale.Category#FORMAT
487     * @return a number format for integer values
488     * @since 1.4

```

```
489     */
490     public final static NumberFormat getIntegerInstance() {
491         return getInstance(Locale.getDefault(Locale.Category.FORMAT), INTEGERSTYLE);
492     }
493
494     /**
495      * Returns an integer number format for the specified locale. The
496      * returned number format is configured to round floating point numbers
497      * to the nearest integer using half-even rounding (see {@link
498      * java.math.RoundingMode#HALF_EVEN RoundingMode.HALF_EVEN}) for formatting,
499      * and to parse only the integer part of an input string (see {@link
500      * #isParseIntegerOnly isParseIntegerOnly}).
501      *
502      * @param inLocale the desired locale
503      * @see #getRoundingMode()
504      * @return a number format for integer values
505      * @since 1.4
506      */
507     public static NumberFormat getIntegerInstance(Locale inLocale) {
508         return getInstance(inLocale, INTEGERSTYLE);
509     }
510
511     /**
512      * Returns a currency format for the current default
513      * {@link java.util.Locale.Category#FORMAT FORMAT} locale.
514      * <p>This is equivalent to calling
515      * {@link #getCurrencyInstance(Locale)
516      * getCurrencyInstance(Locale.getDefault(Locale.Category.FORMAT))}.
517      *
518      * @return the {@code NumberFormat} instance for currency formatting
519      * @see java.util.Locale#getDefault(java.util.Locale.Category)
520      * @see java.util.Locale.Category#FORMAT
521      */
522     public final static NumberFormat getCurrencyInstance() {
523         return getInstance(Locale.getDefault(Locale.Category.FORMAT), CURRENCYSTYLE);
524     }
525
526     /**
527      * Returns a currency format for the specified locale.
528      *
529      * @param inLocale the desired locale
530      * @return the {@code NumberFormat} instance for currency formatting
531      */
532     public static NumberFormat getCurrencyInstance(Locale inLocale) {
533         return getInstance(inLocale, CURRENCYSTYLE);
534     }
535
536     /**
537      * Returns a percentage format for the current default
538      * {@link java.util.Locale.Category#FORMAT FORMAT} locale.
539      * <p>This is equivalent to calling
540      * {@link #getPercentInstance(Locale)
541      * getPercentInstance(Locale.getDefault(Locale.Category.FORMAT))}.
```

```
542     *
543     * @return the {@code NumberFormat} instance for percentage formatting
544     * @see java.util.Locale#getDefault(java.util.Locale.Category)
545     * @see java.util.Locale.Category#FORMAT
546     */
547     public final static NumberFormat getPercentInstance() {
548         return getInstance(Locale.getDefault(Locale.Category.FORMAT), PERCENTSTYLE);
549     }
550
551     /**
552     * Returns a percentage format for the specified locale.
553     *
554     * @param inLocale the desired locale
555     * @return the {@code NumberFormat} instance for percentage formatting
556     */
557     public static NumberFormat getPercentInstance(Locale inLocale) {
558         return getInstance(inLocale, PERCENTSTYLE);
559     }
560
561     /**
562     * Returns a scientific format for the current default locale.
563     */
564     /*public*/ final static NumberFormat getScientificInstance() {
565         return getInstance(Locale.getDefault(Locale.Category.FORMAT), SCIENTIFICSTYLE);
566     }
567
568     /**
569     * Returns a scientific format for the specified locale.
570     *
571     * @param inLocale the desired locale
572     */
573     /*public*/ static NumberFormat getScientificInstance(Locale inLocale) {
574         return getInstance(inLocale, SCIENTIFICSTYLE);
575     }
576
577     /**
578     * Returns an array of all locales for which the
579     * <code>get*Instance</code> methods of this class can return
580     * localized instances.
581     * The returned array represents the union of locales supported by the Java
582     * runtime and by installed
583     * {@link java.text.spi.NumberFormatProvider NumberFormatProvider} implementations.
584     * It must contain at least a <code>Locale</code> instance equal to
585     * {@link java.util.Locale#US Locale.US}.
586     *
587     * @return An array of locales for which localized
588     *         <code>NumberFormat</code> instances are available.
589     */
590     public static Locale[] getAvailableLocales() {
591         LocaleServiceProviderPool pool =
592             LocaleServiceProviderPool.getPool(NumberFormatProvider.class);
593         return pool.getAvailableLocales();
594     }
```

```
595
596 /**
597  * Overrides hashCode.
598  */
599 @Override
600 public int hashCode() {
601     return maximumIntegerDigits * 37 + maxFractionDigits;
602     // just enough fields for a reasonable distribution
603 }
604
605 /**
606  * Overrides equals.
607  */
608 @Override
609 public boolean equals(Object obj) {
610     if (obj == null) {
611         return false;
612     }
613     if (this == obj) {
614         return true;
615     }
616     if (getClass() != obj.getClass()) {
617         return false;
618     }
619     NumberFormat other = (NumberFormat) obj;
620     return (maximumIntegerDigits == other.maximumIntegerDigits
621         && minimumIntegerDigits == other.minimumIntegerDigits
622         && maximumFractionDigits == other.maximumFractionDigits
623         && minimumFractionDigits == other.minimumFractionDigits
624         && groupingUsed == other.groupingUsed
625         && parseIntegerOnly == other.parseIntegerOnly);
626 }
627
628 /**
629  * Overrides Cloneable.
630  */
631 @Override
632 public Object clone() {
633     NumberFormat other = (NumberFormat) super.clone();
634     return other;
635 }
636
637 /**
638  * Returns true if grouping is used in this format. For example, in the
639  * English locale, with grouping on, the number 1234567 might be formatted
640  * as "1,234,567". The grouping separator as well as the size of each group
641  * is locale dependant and is determined by sub-classes of NumberFormat.
642  *
643  * @return {@code true} if grouping is used;
644  *         {@code false} otherwise
645  * @see #setGroupingUsed
646  */
647 public boolean isGroupingUsed() {
```

```
648         return groupingUsed;
649     }
650
651     /**
652      * Set whether or not grouping will be used in this format.
653      *
654      * @param newValue {@code true} if grouping is used;
655      *                 {@code false} otherwise
656      * @see #isGroupingUsed
657      */
658     public void setGroupingUsed(boolean newValue) {
659         groupingUsed = newValue;
660     }
661
662     /**
663      * Returns the maximum number of digits allowed in the integer portion of a
664      * number.
665      *
666      * @return the maximum number of digits
667      * @see #setMaximumIntegerDigits
668      */
669     public int getMaximumIntegerDigits() {
670         return maximumIntegerDigits;
671     }
672
673     /**
674      * Sets the maximum number of digits allowed in the integer portion of a
675      * number. maximumIntegerDigits must be >=; minimumIntegerDigits. If the
676      * new value for maximumIntegerDigits is less than the current value
677      * of minimumIntegerDigits, then minimumIntegerDigits will also be set to
678      * the new value.
679      *
680      * @param newValue the maximum number of integer digits to be shown; if
681      * less than zero, then zero is used. The concrete subclass may enforce an
682      * upper limit to this value appropriate to the numeric type being formatted.
683      * @see #getMaximumIntegerDigits
684      */
685     public void setMaximumIntegerDigits(int newValue) {
686         maximumIntegerDigits = Math.max(0,newValue);
687         if (minimumIntegerDigits > maximumIntegerDigits) {
688             minimumIntegerDigits = maximumIntegerDigits;
689         }
690     }
691
692     /**
693      * Returns the minimum number of digits allowed in the integer portion of a
694      * number.
695      *
696      * @return the minimum number of digits
697      * @see #setMinimumIntegerDigits
698      */
699     public int getMinimumIntegerDigits() {
700         return minimumIntegerDigits;
701     }
```

```
701     }
702
703     /**
704      * Sets the minimum number of digits allowed in the integer portion of a
705      * number. minimumIntegerDigits must be <=; maximumIntegerDigits. If the
706      * new value for minimumIntegerDigits exceeds the current value
707      * of maximumIntegerDigits, then maximumIntegerDigits will also be set to
708      * the new value
709      *
710      * @param newValue the minimum number of integer digits to be shown; if
711      * less than zero, then zero is used. The concrete subclass may enforce an
712      * upper limit to this value appropriate to the numeric type being formatted.
713      * @see #getMinimumIntegerDigits
714      */
715     public void setMinimumIntegerDigits(int newValue) {
716         minimumIntegerDigits = Math.max(0,newValue);
717         if (minimumIntegerDigits > maximumIntegerDigits) {
718             maximumIntegerDigits = minimumIntegerDigits;
719         }
720     }
721
722     /**
723      * Returns the maximum number of digits allowed in the fraction portion of a
724      * number.
725      *
726      * @return the maximum number of digits.
727      * @see #setMaximumFractionDigits
728      */
729     public int getMaximumFractionDigits() {
730         return maximumFractionDigits;
731     }
732
733     /**
734      * Sets the maximum number of digits allowed in the fraction portion of a
735      * number. maximumFractionDigits must be >=; minimumFractionDigits. If the
736      * new value for maximumFractionDigits is less than the current value
737      * of minimumFractionDigits, then minimumFractionDigits will also be set to
738      * the new value.
739      *
740      * @param newValue the maximum number of fraction digits to be shown; if
741      * less than zero, then zero is used. The concrete subclass may enforce an
742      * upper limit to this value appropriate to the numeric type being formatted.
743      * @see #getMaximumFractionDigits
744      */
745     public void setMaximumFractionDigits(int newValue) {
746         maximumFractionDigits = Math.max(0,newValue);
747         if (maximumFractionDigits < minimumFractionDigits) {
748             minimumFractionDigits = maximumFractionDigits;
749         }
750     }
751
752     /**
753      * Returns the minimum number of digits allowed in the fraction portion of a
```

```
754     * number.
755     *
756     * @return the minimum number of digits
757     * @see #setMinimumFractionDigits
758     */
759     public int getMinimumFractionDigits() {
760         return minimumFractionDigits;
761     }
762
763     /**
764     * Sets the minimum number of digits allowed in the fraction portion of a
765     * number. minimumFractionDigits must be <=; maximumFractionDigits. If the
766     * new value for minimumFractionDigits exceeds the current value
767     * of maximumFractionDigits, then maximumIntegerDigits will also be set to
768     * the new value
769     *
770     * @param newValue the minimum number of fraction digits to be shown; if
771     * less than zero, then zero is used. The concrete subclass may enforce an
772     * upper limit to this value appropriate to the numeric type being formatted.
773     * @see #getMinimumFractionDigits
774     */
775     public void setMinimumFractionDigits(int newValue) {
776         minimumFractionDigits = Math.max(0,newValue);
777         if (maximumFractionDigits < minimumFractionDigits) {
778             maximumFractionDigits = minimumFractionDigits;
779         }
780     }
781
782     /**
783     * Gets the currency used by this number format when formatting
784     * currency values. The initial value is derived in a locale dependent
785     * way. The returned value may be null if no valid
786     * currency could be determined and no currency has been set using
787     * {@link #setCurrency(java.util.Currency) setCurrency}.
788     * <p>
789     * The default implementation throws
790     * <code>UnsupportedOperationException</code>.
791     *
792     * @return the currency used by this number format, or <code>null</code>
793     * @exception UnsupportedOperationException if the number format class
794     * doesn't implement currency formatting
795     * @since 1.4
796     */
797     public Currency getCurrency() {
798         throw new UnsupportedOperationException();
799     }
800
801     /**
802     * Sets the currency used by this number format when formatting
803     * currency values. This does not update the minimum or maximum
804     * number of fraction digits used by the number format.
805     * <p>
806     * The default implementation throws
```



```

807 * <code>UnsupportedOperationException</code>.
808 *
809 * @param currency the new currency to be used by this number format
810 * @exception UnsupportedOperationException if the number format class
811 * doesn't implement currency formatting
812 * @exception NullPointerException if <code>currency</code> is null
813 * @since 1.4
814 */
815 public void setCurrency(Currency currency) {
816     throw new UnsupportedOperationException();
817 }
818
819 /**
820 * Gets the {@link java.math.RoundingMode} used in this NumberFormat.
821 * The default implementation of this method in NumberFormat
822 * always throws {@link java.lang.UnsupportedOperationException}.
823 * Subclasses which handle different rounding modes should override
824 * this method.
825 *
826 * @exception UnsupportedOperationException The default implementation
827 *     always throws this exception
828 * @return The <code>RoundingMode</code> used for this NumberFormat.
829 * @see #setRoundingMode(RoundingMode)
830 * @since 1.6
831 */
832 public RoundingMode getRoundingMode() {
833     throw new UnsupportedOperationException();
834 }
835
836 /**
837 * Sets the {@link java.math.RoundingMode} used in this NumberFormat.
838 * The default implementation of this method in NumberFormat always
839 * throws {@link java.lang.UnsupportedOperationException}.
840 * Subclasses which handle different rounding modes should override
841 * this method.
842 *
843 * @exception UnsupportedOperationException The default implementation
844 *     always throws this exception
845 * @exception NullPointerException if <code>roundingMode</code> is null
846 * @param roundingMode The <code>RoundingMode</code> to be used
847 * @see #getRoundingMode()
848 * @since 1.6
849 */
850 public void setRoundingMode(RoundingMode roundingMode) {
851     throw new UnsupportedOperationException();
852 }
853
854 // =====privates=====
855
856 private static NumberFormat getInstance(Locale desiredLocale,
857                                         int choice) {
858     LocaleProviderAdapter adapter;
859     adapter = LocaleProviderAdapter.getAdapter(NumberFormatProvider.class,

```

```

860         desiredLocale);
861     NumberFormat numberFormat = getInstance(adapter, desiredLocale, choice);
862     if (numberFormat == null) {
863         numberFormat = getInstance(LocaleProviderAdapter.forJRE(),
864             desiredLocale, choice);
865     }
866     return numberFormat;
867 }
868
869 private static NumberFormat getInstance(LocaleProviderAdapter adapter,
870     Locale locale, int choice) {
871     NumberFormatProvider provider = adapter.getNumberFormatProvider();
872     NumberFormat numberFormat = null;
873     switch (choice) {
874     case NUMBERSTYLE:
875         numberFormat = provider.getNumberInstance(locale);
876         break;
877     case PERCENTSTYLE:
878         numberFormat = provider.getPercentInstance(locale);
879         break;
880     case CURRENCYSTYLE:
881         numberFormat = provider.getCurrencyInstance(locale);
882         break;
883     case INTEGERSTYLE:
884         numberFormat = provider.getIntegerInstance(locale);
885         break;
886     }
887     return numberFormat;
888 }
889
890 /**
891  * First, read in the default serializable data.
892  *
893  * Then, if <code>serialVersionOnStream</code> is less than 1, indicating that
894  * the stream was written by JDK 1.1,
895  * set the <code>int</code> fields such as <code>maximumIntegerDigits</code>
896  * to be equal to the <code>byte</code> fields such as <code>maxIntegerDigits</code>,
897  * since the <code>int</code> fields were not present in JDK 1.1.
898  * Finally, set serialVersionOnStream back to the maximum allowed value so that
899  * default serialization will work properly if this object is streamed out again.
900  *
901  * <p>If <code>minimumIntegerDigits</code> is greater than
902  * <code>maximumIntegerDigits</code> or <code>minimumFractionDigits</code>
903  * is greater than <code>maximumFractionDigits</code>, then the stream data
904  * is invalid and this method throws an <code>InvalidObjectException</code>.
905  * In addition, if any of these values is negative, then this method throws
906  * an <code>InvalidObjectException</code>.
907  *
908  * @since 1.2
909  */
910 private void readObject(ObjectInputStream stream)
911     throws IOException, ClassNotFoundException
912 {

```

```

913     stream.defaultReadObject();
914     if (serialVersionOnStream < 1) {
915         // Didn't have additional int fields, reassign to use them.
916         maximumIntegerDigits = maxIntegerDigits;
917         minimumIntegerDigits = minIntegerDigits;
918         maximumFractionDigits = maxFractionDigits;
919         minimumFractionDigits = minFractionDigits;
920     }
921     if (minimumIntegerDigits > maximumIntegerDigits ||
922         minimumFractionDigits > maximumFractionDigits ||
923         minimumIntegerDigits < 0 || minimumFractionDigits < 0) {
924         throw new InvalidObjectException("Digit count range invalid");
925     }
926     serialVersionOnStream = currentSerialVersion;
927 }
928
929 /**
930  * Write out the default serializable data, after first setting
931  * the <code>byte</code> fields such as <code>maxIntegerDigits</code> to be
932  * equal to the <code>int</code> fields such as <code>maximumIntegerDigits</code>
933  * (or to <code>Byte.MAX_VALUE</code>, whichever is smaller), for compatibility
934  * with the JDK 1.1 version of the stream format.
935  *
936  * @since 1.2
937  */
938 private void writeObject(ObjectOutputStream stream)
939     throws IOException
940 {
941     maxIntegerDigits = (maximumIntegerDigits > Byte.MAX_VALUE) ?
942         Byte.MAX_VALUE : (byte)maximumIntegerDigits;
943     minIntegerDigits = (minimumIntegerDigits > Byte.MAX_VALUE) ?
944         Byte.MAX_VALUE : (byte)minimumIntegerDigits;
945     maxFractionDigits = (maximumFractionDigits > Byte.MAX_VALUE) ?
946         Byte.MAX_VALUE : (byte)maximumFractionDigits;
947     minFractionDigits = (minimumFractionDigits > Byte.MAX_VALUE) ?
948         Byte.MAX_VALUE : (byte)minimumFractionDigits;
949     stream.defaultWriteObject();
950 }
951
952 // Constants used by factory methods to specify a style of format.
953 private static final int NUMBERSTYLE = 0;
954 private static final int CURRENCYSTYLE = 1;
955 private static final int PERCENTSTYLE = 2;
956 private static final int SCIENTIFICSTYLE = 3;
957 private static final int INTEGERSTYLE = 4;
958
959 /**
960  * True if the grouping (i.e. thousands) separator is used when
961  * formatting and parsing numbers.
962  *
963  * @serial
964  * @see #isGroupingUsed
965  */

```

```

966     private boolean groupingUsed = true;
967
968     /**
969      * The maximum number of digits allowed in the integer portion of a
970      * number. maxIntegerDigits must be greater than or equal to
971      * minIntegerDigits.
972      * <p>
973      * <strong>Note:</strong> This field exists only for serialization
974      * compatibility with JDK 1.1. In Java platform 2 v1.2 and higher, the new
975      * int field maximumIntegerDigits is used instead.
976      * When writing to a stream, maxIntegerDigits is set to
977      * maximumIntegerDigits or Byte.MAX_VALUE,
978      * whichever is smaller. When reading from a stream, this field is used
979      * only if serialVersionOnStream is less than 1.
980      *
981      * @serial
982      * @see #getMaximumIntegerDigits
983      */
984     private byte    maxIntegerDigits = 40;
985
986     /**
987      * The minimum number of digits allowed in the integer portion of a
988      * number. minimumIntegerDigits must be less than or equal to
989      * maximumIntegerDigits.
990      * <p>
991      * <strong>Note:</strong> This field exists only for serialization
992      * compatibility with JDK 1.1. In Java platform 2 v1.2 and higher, the new
993      * int field minimumIntegerDigits is used instead.
994      * When writing to a stream, minIntegerDigits is set to
995      * minimumIntegerDigits or Byte.MAX_VALUE,
996      * whichever is smaller. When reading from a stream, this field is used
997      * only if serialVersionOnStream is less than 1.
998      *
999      * @serial
1000     * @see #getMinimumIntegerDigits
1001     */
1002     private byte    minIntegerDigits = 1;
1003
1004     /**
1005      * The maximum number of digits allowed in the fractional portion of a
1006      * number. maximumFractionDigits must be greater than or equal to
1007      * minimumFractionDigits.
1008      * <p>
1009      * <strong>Note:</strong> This field exists only for serialization
1010      * compatibility with JDK 1.1. In Java platform 2 v1.2 and higher, the new
1011      * int field maximumFractionDigits is used instead.
1012      * When writing to a stream, maxFractionDigits is set to
1013      * maximumFractionDigits or Byte.MAX_VALUE,
1014      * whichever is smaller. When reading from a stream, this field is used
1015      * only if serialVersionOnStream is less than 1.
1016      *
1017      * @serial
1018      * @see #getMaximumFractionDigits

```

```
1019     */
1020     private byte    maxFractionDigits = 3;    // invariant, >= minFractionDigits
1021
1022     /**
1023      * The minimum number of digits allowed in the fractional portion of a
1024      * number. <code>minimumFractionDigits</code> must be less than or equal to
1025      * <code>maximumFractionDigits</code>.
1026      * <p>
1027      * <strong>Note:</strong> This field exists only for serialization
1028      * compatibility with JDK 1.1. In Java platform 2 v1.2 and higher, the new
1029      * <code>int</code> field <code>minimumFractionDigits</code> is used instead.
1030      * When writing to a stream, <code>minFractionDigits</code> is set to
1031      * <code>minimumFractionDigits</code> or <code>Byte.MAX_VALUE</code>,
1032      * whichever is smaller. When reading from a stream, this field is used
1033      * only if <code>serialVersionOnStream</code> is less than 1.
1034      *
1035      * @serial
1036      * @see #getMinimumFractionDigits
1037      */
1038     private byte    minFractionDigits = 0;
1039
1040     /**
1041      * True if this format will parse numbers as integers only.
1042      *
1043      * @serial
1044      * @see #isParseIntegerOnly
1045      */
1046     private boolean parseIntegerOnly = false;
1047
1048     // new fields for 1.2. byte is too small for integer digits.
1049
1050     /**
1051      * The maximum number of digits allowed in the integer portion of a
1052      * number. <code>maximumIntegerDigits</code> must be greater than or equal to
1053      * <code>minimumIntegerDigits</code>.
1054      *
1055      * @serial
1056      * @since 1.2
1057      * @see #getMaximumIntegerDigits
1058      */
1059     private int     maximumIntegerDigits = 40;
1060
1061     /**
1062      * The minimum number of digits allowed in the integer portion of a
1063      * number. <code>minimumIntegerDigits</code> must be less than or equal to
1064      * <code>maximumIntegerDigits</code>.
1065      *
1066      * @serial
1067      * @since 1.2
1068      * @see #getMinimumIntegerDigits
1069      */
1070     private int     minimumIntegerDigits = 1;
1071
```

```

1072 /**
1073  * The maximum number of digits allowed in the fractional portion of a
1074  * number. <code>maximumFractionDigits</code> must be greater than or equal to
1075  * <code>minimumFractionDigits</code>.
1076  *
1077  * @serial
1078  * @since 1.2
1079  * @see #getMaximumFractionDigits
1080  */
1081 private int    maximumFractionDigits = 3;    // invariant, >= minFractionDigits
1082
1083 /**
1084  * The minimum number of digits allowed in the fractional portion of a
1085  * number. <code>minimumFractionDigits</code> must be less than or equal to
1086  * <code>maximumFractionDigits</code>.
1087  *
1088  * @serial
1089  * @since 1.2
1090  * @see #getMinimumFractionDigits
1091  */
1092 private int    minimumFractionDigits = 0;
1093
1094 static final int currentSerialVersion = 1;
1095
1096 /**
1097  * Describes the version of <code>NumberFormat</code> present on the stream.
1098  * Possible values are:
1099  * <ul>
1100  * <li><b>0</b> (or uninitialized): the JDK 1.1 version of the stream format.
1101  *     In this version, the <code>int</code> fields such as
1102  *     <code>maximumIntegerDigits</code> were not present, and the <code>byte</code>
1103  *     fields such as <code>maxIntegerDigits</code> are used instead.
1104  *
1105  * <li><b>1</b>: the 1.2 version of the stream format. The values of the
1106  *     <code>byte</code> fields such as <code>maxIntegerDigits</code> are ignored,
1107  *     and the <code>int</code> fields such as <code>maximumIntegerDigits</code>
1108  *     are used instead.
1109  * </ul>
1110  * When streaming out a <code>NumberFormat</code>, the most recent format
1111  * (corresponding to the highest allowable <code>serialVersionOnStream</code>)
1112  * is always written.
1113  *
1114  * @serial
1115  * @since 1.2
1116  */
1117 private int serialVersionOnStream = currentSerialVersion;
1118
1119 // Removed "implements Cloneable" clause. Needs to update serialization
1120 // ID for backward compatibility.
1121 static final long serialVersionUID = -2308460125733713944L;
1122
1123
1124 //

```

```

1125 // class for AttributedCharacterIterator attributes
1126 //
1127 /**
1128  * Defines constants that are used as attribute keys in the
1129  * <code>AttributedCharacterIterator</code> returned
1130  * from <code>NumberFormat.formatToCharacterIterator</code> and as
1131  * field identifiers in <code>FieldPosition</code>.
1132  *
1133  * @since 1.4
1134  */
1135 public static class Field extends Format.Field {
1136
1137     // Proclaim serial compatibility with 1.4 FCS
1138     private static final long serialVersionUID = 7494728892700160890L;
1139
1140     // table of all instances in this class, used by readResolve
1141     private static final Map<String, Field> instanceMap = new HashMap<>(11);
1142
1143     /**
1144      * Creates a Field instance with the specified
1145      * name.
1146      *
1147      * @param name Name of the attribute
1148      */
1149     protected Field(String name) {
1150         super(name);
1151         if (this.getClass() == NumberFormat.Field.class) {
1152             instanceMap.put(name, this);
1153         }
1154     }
1155
1156     /**
1157      * Resolves instances being deserialized to the predefined constants.
1158      *
1159      * @throws InvalidObjectException if the constant could not be resolved.
1160      * @return resolved NumberFormat.Field constant
1161      */
1162     @Override
1163     protected Object readResolve() throws InvalidObjectException {
1164         if (this.getClass() != NumberFormat.Field.class) {
1165             throw new InvalidObjectException("subclass didn't correctly implement readResolve")
1166                 ;
1167         }
1168
1169         Object instance = instanceMap.get(getName());
1170         if (instance != null) {
1171             return instance;
1172         } else {
1173             throw new InvalidObjectException("unknown attribute name");
1174         }
1175     }
1176
1177     /**

```

```
1177     * Constant identifying the integer field.
1178     */
1179     public static final Field INTEGER = new Field("integer");
1180
1181     /**
1182     * Constant identifying the fraction field.
1183     */
1184     public static final Field FRACTION = new Field("fraction");
1185
1186     /**
1187     * Constant identifying the exponent field.
1188     */
1189     public static final Field EXPONENT = new Field("exponent");
1190
1191     /**
1192     * Constant identifying the decimal separator field.
1193     */
1194     public static final Field DECIMAL_SEPARATOR =
1195         new Field("decimal separator");
1196
1197     /**
1198     * Constant identifying the sign field.
1199     */
1200     public static final Field SIGN = new Field("sign");
1201
1202     /**
1203     * Constant identifying the grouping separator field.
1204     */
1205     public static final Field GROUPING_SEPARATOR =
1206         new Field("grouping separator");
1207
1208     /**
1209     * Constant identifying the exponent symbol field.
1210     */
1211     public static final Field EXPONENT_SYMBOL = new
1212         Field("exponent symbol");
1213
1214     /**
1215     * Constant identifying the percent field.
1216     */
1217     public static final Field PERCENT = new Field("percent");
1218
1219     /**
1220     * Constant identifying the permille field.
1221     */
1222     public static final Field PERMILLE = new Field("per mille");
1223
1224     /**
1225     * Constant identifying the currency field.
1226     */
1227     public static final Field CURRENCY = new Field("currency");
1228
1229     /**
```



```

1230     * Constant identifying the exponent sign field.
1231     */
1232     public static final Field EXPONENT_SIGN = new Field("exponent sign");
1233 }
1234 }

```

2.26 ParseException.java

Secuencia 35: java/text/ParseException.java

```

1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28 * (C) Copyright IBM Corp. 1996 - 1998 - All Rights Reserved
29 *
30 * The original version of this source code and documentation is copyrighted
31 * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32 * materials are provided under terms of a License Agreement between Taligent
33 * and Sun. This technology is protected by multiple US and International
34 * patents. This notice and attribution to Taligent may not be removed.
35 * Taligent is a registered trademark of Taligent, Inc.
36 *
37 */
38
39 package java.text;
40
41 /**
42 * Signals that an error has been reached unexpectedly
43 * while parsing.
44 * @see java.lang.Exception

```

```

45  * @see java.text.Format
46  * @see java.text.FieldPosition
47  * @author      Mark Davis
48  */
49  public
50  class ParseException extends Exception {
51
52      private static final long serialVersionUID = 2703218443322787634L;
53
54      /**
55       * Constructs a ParseException with the specified detail message and
56       * offset.
57       * A detail message is a String that describes this particular exception.
58       *
59       * @param s the detail message
60       * @param errorOffset the position where the error is found while parsing.
61       */
62      public ParseException(String s, int errorOffset) {
63          super(s);
64          this.errorOffset = errorOffset;
65      }
66
67      /**
68       * Returns the position where the error was found.
69       *
70       * @return the position where the error was found
71       */
72      public int getErrorOffset () {
73          return errorOffset;
74      }
75
76      //===== privates =====
77      /**
78       * The zero-based character offset into the string being parsed at which
79       * the error was found during parsing.
80       * @serial
81       */
82      private int errorOffset;
83  }

```

2.27 ParsePosition.java

Secuencia 36: java/text/ParsePosition.java

```

1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *

```

```
11  *
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28  * (C) Copyright IBM Corp. 1996 - 1998 - All Rights Reserved
29  *
30  * The original version of this source code and documentation is copyrighted
31  * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32  * materials are provided under terms of a License Agreement between Taligent
33  * and Sun. This technology is protected by multiple US and International
34  * patents. This notice and attribution to Taligent may not be removed.
35  * Taligent is a registered trademark of Taligent, Inc.
36  *
37  */
38
39 package java.text;
40
41
42 /**
43  * <code>ParsePosition</code> is a simple class used by <code>Format</code>
44  * and its subclasses to keep track of the current position during parsing.
45  * The <code>parseObject</code> method in the various <code>Format</code>
46  * classes requires a <code>ParsePosition</code> object as an argument.
47  *
48  * <p>
49  * By design, as you parse through a string with different formats,
50  * you can use the same <code>ParsePosition</code>, since the index parameter
51  * records the current position.
52  *
53  * @author      Mark Davis
54  * @see         java.text.Format
55  */
56
57 public class ParsePosition {
58
59     /**
60      * Input: the place you start parsing.
61      * <br>Output: position where the parse stopped.
62      * This is designed to be used serially,
63      * with each call setting index up for the next one.
```

```
64     */
65     int index = 0;
66     int errorIndex = -1;
67
68     /**
69      * Retrieve the current parse position. On input to a parse method, this
70      * is the index of the character at which parsing will begin; on output, it
71      * is the index of the character following the last character parsed.
72      *
73      * @return the current parse position
74      */
75     public int getIndex() {
76         return index;
77     }
78
79     /**
80      * Set the current parse position.
81      *
82      * @param index the current parse position
83      */
84     public void setIndex(int index) {
85         this.index = index;
86     }
87
88     /**
89      * Create a new ParsePosition with the given initial index.
90      *
91      * @param index initial index
92      */
93     public ParsePosition(int index) {
94         this.index = index;
95     }
96
97     /**
98      * Set the index at which a parse error occurred. Formatters
99      * should set this before returning an error code from their
100     * parseObject method. The default value is -1 if this is not set.
101     *
102     * @param ei the index at which an error occurred
103     * @since 1.2
104     */
105     public void setErrorIndex(int ei)
106     {
107         errorIndex = ei;
108     }
109
110     /**
111      * Retrieve the index at which an error occurred, or -1 if the
112      * error index has not been set.
113      *
114      * @return the index at which an error occurred
115      * @since 1.2
116      */
117     public int getErrorIndex()
```

```

117     {
118         return errorIndex;
119     }
120
121     /**
122      * Overrides equals
123      */
124     public boolean equals(Object obj)
125     {
126         if (obj == null) return false;
127         if (!(obj instanceof ParsePosition))
128             return false;
129         ParsePosition other = (ParsePosition) obj;
130         return (index == other.index && errorIndex == other.errorIndex);
131     }
132
133     /**
134      * Returns a hash code for this ParsePosition.
135      * @return a hash code value for this object
136      */
137     public int hashCode() {
138         return (errorIndex << 16) | index;
139     }
140
141     /**
142      * Return a string representation of this ParsePosition.
143      * @return a string representation of this object
144      */
145     public String toString() {
146         return getClass().getName() +
147             "[index=" + index +
148             ",errorIndex=" + errorIndex + ']';
149     }
150 }

```

2.28 PatternEntry.java

Secuencia 37: java/text/PatternEntry.java

```

1  /*
2  * Copyright (c) 1996, 2000, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *

```

```
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28  * (C) Copyright IBM Corp. 1996, 1997 - All Rights Reserved
29  *
30  * The original version of this source code and documentation is copyrighted
31  * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32  * materials are provided under terms of a License Agreement between Taligent
33  * and Sun. This technology is protected by multiple US and International
34  * patents. This notice and attribution to Taligent may not be removed.
35  * Taligent is a registered trademark of Taligent, Inc.
36  *
37  */
38
39  package java.text;
40
41  import java.lang.Character;
42
43  /**
44   * Utility class for normalizing and merging patterns for collation.
45   * This is to be used with MergeCollation for adding patterns to an
46   * existing rule table.
47   * @see      MergeCollation
48   * @author    Mark Davis, Helena Shih
49   */
50
51  class PatternEntry {
52      /**
53       * Gets the current extension, quoted
54       */
55      public void appendQuotedExtension(StringBuffer toAddTo) {
56          appendQuoted(extension,toAddTo);
57      }
58
59      /**
60       * Gets the current chars, quoted
61       */
62      public void appendQuotedChars(StringBuffer toAddTo) {
63          appendQuoted(chars,toAddTo);
64      }
65
66      /**
67       * WARNING this is used for searching in a Vector.
68       * Because Vector.indexOf doesn't take a comparator,
```

```
69     * this method is ill-defined and ignores strength.
70     */
71     public boolean equals(Object obj) {
72         if (obj == null) return false;
73         PatternEntry other = (PatternEntry) obj;
74         boolean result = chars.equals(other.chars);
75         return result;
76     }
77
78     public int hashCode() {
79         return chars.hashCode();
80     }
81
82     /**
83      * For debugging.
84      */
85     public String toString() {
86         StringBuffer result = new StringBuffer();
87         addToBuffer(result, true, false, null);
88         return result.toString();
89     }
90
91     /**
92      * Gets the strength of the entry.
93      */
94     final int getStrength() {
95         return strength;
96     }
97
98     /**
99      * Gets the expanding characters of the entry.
100     */
101     final String getExtension() {
102         return extension;
103     }
104
105     /**
106      * Gets the core characters of the entry.
107     */
108     final String getChars() {
109         return chars;
110     }
111
112     // ===== privates =====
113
114     void addToBuffer(StringBuffer toAddTo,
115                     boolean showExtension,
116                     boolean showWhiteSpace,
117                     PatternEntry lastEntry)
118     {
119         if (showWhiteSpace && toAddTo.length() > 0)
120             if (strength == Collator.PRIMARY || lastEntry != null)
121                 toAddTo.append('\n');
```

```

122         else
123             toAddTo.append(' ');
124     if (lastEntry != null) {
125         toAddTo.append('&');
126         if (showWhiteSpace)
127             toAddTo.append(' ');
128         lastEntry.appendQuotedChars(toAddTo);
129         appendQuotedExtension(toAddTo);
130         if (showWhiteSpace)
131             toAddTo.append(' ');
132     }
133     switch (strength) {
134     case Collator.IDENTICAL: toAddTo.append('='); break;
135     case Collator.TERTIARY:  toAddTo.append(','); break;
136     case Collator.SECONDARY: toAddTo.append(';'); break;
137     case Collator.PRIMARY:   toAddTo.append('<'); break;
138     case RESET:              toAddTo.append('&'); break;
139     case UNSET:              toAddTo.append('?'); break;
140     }
141     if (showWhiteSpace)
142         toAddTo.append(' ');
143     appendQuoted(chars,toAddTo);
144     if (showExtension && extension.length() != 0) {
145         toAddTo.append('/');
146         appendQuoted(extension,toAddTo);
147     }
148 }
149
150 static void appendQuoted(String chars, StringBuffer toAddTo) {
151     boolean inQuote = false;
152     char ch = chars.charAt(0);
153     if (Character.isSpaceChar(ch)) {
154         inQuote = true;
155         toAddTo.append('\\');
156     } else {
157         if (PatternEntry.isSpecialChar(ch)) {
158             inQuote = true;
159             toAddTo.append('\\');
160         } else {
161             switch (ch) {
162             case 0x0010: case '\\f': case '\\r':
163             case '\\t': case '\\n': case '@':
164                 inQuote = true;
165                 toAddTo.append('\\');
166                 break;
167             case '\\':
168                 inQuote = true;
169                 toAddTo.append('\\');
170                 break;
171             default:
172                 if (inQuote) {
173                     inQuote = false; toAddTo.append('\\');
174                 }

```



```

175         break;
176     }
177 }
178 }
179 addTo.append(chars);
180 if (inQuote)
181     addTo.append('\');
182 }
183
184 //=====
185 // Parsing a pattern into a list of PatternEntries....
186 //=====
187
188 PatternEntry(int strength,
189             StringBuffer chars,
190             StringBuffer extension)
191 {
192     this.strength = strength;
193     this.chars = chars.toString();
194     this.extension = (extension.length() > 0) ? extension.toString()
195                                     : "";
196 }
197
198 static class Parser {
199     private String pattern;
200     private int i;
201
202     public Parser(String pattern) {
203         this.pattern = pattern;
204         this.i = 0;
205     }
206
207     public PatternEntry next() throws ParseException {
208         int newStrength = UNSET;
209
210         newChars.setLength(0);
211         newExtension.setLength(0);
212
213         boolean inChars = true;
214         boolean inQuote = false;
215     mainLoop:
216         while (i < pattern.length()) {
217             char ch = pattern.charAt(i);
218             if (inQuote) {
219                 if (ch == '\') {
220                     inQuote = false;
221                 } else {
222                     if (newChars.length() == 0) newChars.append(ch);
223                     else if (inChars) newChars.append(ch);
224                     else newExtension.append(ch);
225                 }
226             } else switch (ch) {
227                 case '=': if (newStrength != UNSET) break mainLoop;

```

```

228         newStrength = Collator.IDENTICAL; break;
229     case ',': if (newStrength != UNSET) break mainLoop;
230         newStrength = Collator.TERTIARY; break;
231     case ';': if (newStrength != UNSET) break mainLoop;
232         newStrength = Collator.SECONDARY; break;
233     case '<': if (newStrength != UNSET) break mainLoop;
234         newStrength = Collator.PRIMARY; break;
235     case '&': if (newStrength != UNSET) break mainLoop;
236         newStrength = RESET; break;
237     case '\t':
238     case '\n':
239     case '\f':
240     case '\r':
241     case ' ': break; // skip whitespace TODO use Character
242     case '/': inChars = false; break;
243     case '\':
244         inQuote = true;
245         ch = pattern.charAt(++i);
246         if (newChars.length() == 0) newChars.append(ch);
247         else if (inChars) newChars.append(ch);
248         else newExtension.append(ch);
249         break;
250     default:
251         if (newStrength == UNSET) {
252             throw new ParseException
253                 ("missing char (,;<&) : " +
254                 pattern.substring(i,
255                     (i+10 < pattern.length()) ?
256                     i+10 : pattern.length()),
257                 i);
258         }
259         if (PatternEntry.isSpecialChar(ch) && (inQuote == false))
260             throw new ParseException
261                 ("Unquoted punctuation character : " + Integer.toString(ch, 16), i);
262         if (inChars) {
263             newChars.append(ch);
264         } else {
265             newExtension.append(ch);
266         }
267         break;
268     }
269     i++;
270 }
271 if (newStrength == UNSET)
272     return null;
273 if (newChars.length() == 0) {
274     throw new ParseException
275         ("missing chars (,;<&): " +
276         pattern.substring(i,
277             (i+10 < pattern.length()) ?
278             i+10 : pattern.length()),
279         i);
280 }

```

```

281         return new PatternEntry(newStrength, newChars, newExtension);
282     }
283
284     // We re-use these objects in order to improve performance
285     private StringBuffer newChars = new StringBuffer();
286     private StringBuffer newExtension = new StringBuffer();
287
288 }
289
290 static boolean isSpecialChar(char ch) {
291     return ((ch == '\u0020') ||
292         ((ch <= '\u002F') && (ch >= '\u0022')) ||
293         ((ch <= '\u003F') && (ch >= '\u003A')) ||
294         ((ch <= '\u0060') && (ch >= '\u005B')) ||
295         ((ch <= '\u007E') && (ch >= '\u007B')));
296 }
297
298
299 static final int RESET = -2;
300 static final int UNSET = -1;
301
302 int strength = UNSET;
303 String chars = "";
304 String extension = "";
305 }

```

2.29 RBCollationTables.java

Secuencia 38: java/text/RBCollationTables.java

```

1  /*
2  * Copyright (c) 1999, 2012, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *

```

```

24  */
25
26  /*
27   * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28   * (C) Copyright IBM Corp. 1996-1998 - All Rights Reserved
29   *
30   * The original version of this source code and documentation is copyrighted
31   * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32   * materials are provided under terms of a License Agreement between Taligent
33   * and Sun. This technology is protected by multiple US and International
34   * patents. This notice and attribution to Taligent may not be removed.
35   * Taligent is a registered trademark of Taligent, Inc.
36   *
37   */
38
39  package java.text;
40
41  import java.util.Vector;
42  import sun.text.UCompactIntArray;
43  import sun.text.IntHashtable;
44
45  /**
46   * This class contains the static state of a RuleBasedCollator: The various
47   * tables that are used by the collation routines. Several RuleBasedCollators
48   * can share a single RBCollationTables object, easing memory requirements and
49   * improving performance.
50   */
51  final class RBCollationTables {
52      //=====
53      // The following diagram shows the data structure of the RBCollationTables object.
54      // Suppose we have the rule, where 'o-umlaut' is the unicode char 0x00F6.
55      // "a, A < b, B < c, C, ch, cH, Ch, CH < d, D ... < o, O; 'o-umlaut'/E, 'O-umlaut'/E ...".
56      // What the rule says is, sorts 'ch'ligatures and 'c' only with tertiary difference and
57      // sorts 'o-umlaut' as if it's always expanded with 'e'.
58      //
59      // mapping table                contracting list                expanding list
60      // (contains all unicode char
61      // entries)
62      //
63      //      +>|_*_|->| 'c' | v('c') | +>|v('o')|v('umlaut')|v('e')|
64      // |_\u0001_|-> v('\u0001') | |_:| |-----| | |-----|
65      // |_\u0002_|-> v('\u0002') | |_:| | 'ch' | v('ch') | | | : |
66      // |____:____| | |_:| |-----| | |-----|
67      // |____:____| | | | | 'cH' | v('cH') | | | : |
68      // |__'a'____|-> v('a') | | | | |-----| | |-----|
69      // |__'b'____|-> v('b') | | | | | 'Ch' | v('Ch') | | | : |
70      // |____:____| | | | | |-----| | |-----|
71      // |____:____| | | | | | 'CH' | v('CH') | | | : |
72      // |____:____| | | | | |-----| | |-----|
73      // |o-umlaut|-----| | | | | : |
74      // |____:____|
75      //
76      // Noted by Helena Shih on 6/23/97

```

```

77 //=====
78
79 public RBCollationTables(String rules, int decmp) throws ParseException {
80     this.rules = rules;
81
82     RTableBuilder builder = new RTableBuilder(new BuildAPI());
83     builder.build(rules, decmp); // this object is filled in through
84                                 // the BuildAPI object
85 }
86
87 final class BuildAPI {
88     /**
89      * Private constructor. Prevents anyone else besides RTableBuilder
90      * from gaining direct access to the internals of this class.
91      */
92     private BuildAPI() {
93     }
94
95     /**
96      * This function is used by RTableBuilder to fill in all the members of this
97      * object. (Effectively, the builder class functions as a "friend" of this
98      * class, but to avoid changing too much of the logic, it carries around "shadow"
99      * copies of all these variables until the end of the build process and then
100     * copies them en masse into the actual tables object once all the construction
101     * logic is complete. This function does that "copying en masse".
102     * @param f2ary The value for frenchSec (the French-secondary flag)
103     * @param swap The value for SE Asian swapping rule
104     * @param map The collator's character-mapping table (the value for mapping)
105     * @param cTbl The collator's contracting-character table (the value for contractTable)
106     * @param eTbl The collator's expanding-character table (the value for expandTable)
107     * @param cFlgs The hash table of characters that participate in contracting-
108     *               character sequences (the value for contractFlags)
109     * @param mso The value for maxSecOrder
110     * @param mto The value for maxTerOrder
111     */
112     void fillInTables(boolean f2ary,
113                      boolean swap,
114                      UCompactIntArray map,
115                      Vector<Vector<EntryPair>> cTbl,
116                      Vector<int[]> eTbl,
117                      IntHashtable cFlgs,
118                      short mso,
119                      short mto) {
120         frenchSec = f2ary;
121         seAsianSwapping = swap;
122         mapping = map;
123         contractTable = cTbl;
124         expandTable = eTbl;
125         contractFlags = cFlgs;
126         maxSecOrder = mso;
127         maxTerOrder = mto;
128     }
129 }

```

```

130
131 /**
132  * Gets the table-based rules for the collation object.
133  * @return returns the collation rules that the table collation object
134  * was created from.
135  */
136 public String getRules()
137 {
138     return rules;
139 }
140
141 public boolean isFrenchSec() {
142     return frenchSec;
143 }
144
145 public boolean isSEAsianSwapping() {
146     return seAsianSwapping;
147 }
148
149 // =====
150 // internal (for use by CollationElementIterator)
151 // =====
152
153 /**
154  * Get the entry of hash table of the contracting string in the collation
155  * table.
156  * @param ch the starting character of the contracting string
157  */
158 Vector<EntryPair> getContractValues(int ch)
159 {
160     int index = mapping.elementAt(ch);
161     return getContractValuesImpl(index - CONTRACTCHARINDEX);
162 }
163
164 //get contract values from contractTable by index
165 private Vector<EntryPair> getContractValuesImpl(int index)
166 {
167     if (index >= 0)
168     {
169         return contractTable.elementAt(index);
170     }
171     else // not found
172     {
173         return null;
174     }
175 }
176
177 /**
178  * Returns true if this character appears anywhere in a contracting
179  * character sequence. (Used by CollationElementIterator.setOffset().)
180  */
181 boolean usedInContractSeq(int c) {
182     return contractFlags.get(c) == 1;

```

```

183     }
184
185     /**
186      * Return the maximum length of any expansion sequences that end
187      * with the specified comparison order.
188      *
189      * @param order a collation order returned by previous or next.
190      * @return the maximum length of any expansion sequences ending
191      *         with the specified order.
192      *
193      * @see CollationElementIterator#getMaxExpansion
194      */
195     int getMaxExpansion(int order) {
196         int result = 1;
197
198         if (expandTable != null) {
199             // Right now this does a linear search through the entire
200             // expansion table. If a collator had a large number of expansions,
201             // this could cause a performance problem, but in practise that
202             // rarely happens
203             for (int i = 0; i < expandTable.size(); i++) {
204                 int[] valueList = expandTable.elementAt(i);
205                 int length = valueList.length;
206
207                 if (length > result && valueList[length-1] == order) {
208                     result = length;
209                 }
210             }
211         }
212
213         return result;
214     }
215
216     /**
217      * Get the entry of hash table of the expanding string in the collation
218      * table.
219      * @param idx the index of the expanding string value list
220      */
221     final int[] getExpandValueList(int idx) {
222         return expandTable.elementAt(idx - EXPANDCHARINDEX);
223     }
224
225     /**
226      * Get the comparison order of a character from the collation table.
227      * @return the comparison order of a character.
228      */
229     int getUnicodeOrder(int ch) {
230         return mapping.elementAt(ch);
231     }
232
233     short getMaxSecOrder() {
234         return maxSecOrder;
235     }

```

```

236
237     short getMaxTerOrder() {
238         return maxTerOrder;
239     }
240
241     /**
242      * Reverse a string.
243      */
244     //shemran/Note: this is used for secondary order value reverse, no
245     //                need to consider supplementary pair.
246     static void reverse (StringBuffer result, int from, int to)
247     {
248         int i = from;
249         char swap;
250
251         int j = to - 1;
252         while (i < j) {
253             swap = result.charAt(i);
254             result.setCharAt(i, result.charAt(j));
255             result.setCharAt(j, swap);
256             i++;
257             j--;
258         }
259     }
260
261     final static int getEntry(Vector<EntryPair> list, String name, boolean fwd) {
262         for (int i = 0; i < list.size(); i++) {
263             EntryPair pair = list.elementAt(i);
264             if (pair.fwd == fwd && pair.entryName.equals(name)) {
265                 return i;
266             }
267         }
268         return UNMAPPED;
269     }
270
271     // =====
272     // constants
273     // =====
274     //sherman/ToDo: is the value big enough????
275     final static int EXPANDCHARINDEX = 0x7E000000; // Expand index follows
276     final static int CONTRACTCHARINDEX = 0x7F000000; // contract indexes follow
277     final static int UNMAPPED = 0xFFFFFFFF;
278
279     final static int PRIMARYORDERMASK = 0xffff0000;
280     final static int SECONDARYORDERMASK = 0x0000ff00;
281     final static int TERTIARYORDERMASK = 0x000000ff;
282     final static int PRIMARYDIFFERENCEONLY = 0xffff0000;
283     final static int SECONDARYDIFFERENCEONLY = 0xffffff00;
284     final static int PRIMARYORDERSHIFT = 16;
285     final static int SECONDARYORDERSHIFT = 8;
286
287     // =====
288     // instance variables

```



```

289 // =====
290 private String rules = null;
291 private boolean frenchSec = false;
292 private boolean seAsianSwapping = false;
293
294 private UCompactIntArray mapping = null;
295 private Vector<Vector<EntryPair>> contractTable = null;
296 private Vector<int[]> expandTable = null;
297 private IntHashtable contractFlags = null;
298
299 private short maxSecOrder = 0;
300 private short maxTerOrder = 0;
301 }

```

2.30 RTableBuilder.java

Secuencia 39: java/text/RTableBuilder.java

```

1  /*
2  * Copyright (c) 1999, 2012, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28 * (C) Copyright IBM Corp. 1996-1998 - All Rights Reserved
29 *
30 * The original version of this source code and documentation is copyrighted
31 * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32 * materials are provided under terms of a License Agreement between Taligent
33 * and Sun. This technology is protected by multiple US and International
34 * patents. This notice and attribution to Taligent may not be removed.
35 * Taligent is a registered trademark of Taligent, Inc.
36 *

```

```

37  */
38
39  package java.text;
40
41  import java.util.Vector;
42  import sun.text.UCompactIntArray;
43  import sun.text.IntHashtable;
44  import sun.text.ComposedCharIter;
45  import sun.text.CollatorUtilities;
46  import sun.text.normalizer.NormalizerImpl;
47
48  /**
49   * This class contains all the code to parse a RuleBasedCollator pattern
50   * and build a RBCollationTables object from it. A particular instance
51   * of this class exists only during the actual build process-- once an
52   * RBCollationTables object has been built, the RTableBuilder object
53   * goes away. This object carries all of the state which is only needed
54   * during the build process, plus a "shadow" copy of all of the state
55   * that will go into the tables object itself. This object communicates
56   * with RBCollationTables through a separate class, RBCollationTables.BuildAPI,
57   * this is an inner class of RBCollationTables and provides a separate
58   * private API for communication with RTableBuilder.
59   * This class isn't just an inner class of RBCollationTables itself because
60   * of its large size. For source-code readability, it seemed better for the
61   * builder to have its own source file.
62   */
63  final class RTableBuilder {
64
65      public RTableBuilder(RBCollationTables.BuildAPI tables) {
66          this.tables = tables;
67      }
68
69      /**
70       * Create a table-based collation object with the given rules.
71       * This is the main function that actually builds the tables and
72       * stores them back in the RBCollationTables object. It is called
73       * ONLY by the RBCollationTables constructor.
74       * @see RuleBasedCollator#RuleBasedCollator
75       * @exception ParseException If the rules format is incorrect.
76       */
77
78      public void build(String pattern, int decmp) throws ParseException
79      {
80          boolean isSource = true;
81          int i = 0;
82          String expChars;
83          String groupChars;
84          if (pattern.length() == 0)
85              throw new ParseException("Build rules empty.", 0);
86
87          // This array maps Unicode characters to their collation ordering
88          mapping = new UCompactIntArray(RBCollationTables.UNMAPPED);
89          // Normalize the build rules. Find occurrences of all decomposed characters

```

```

90     // and normalize the rules before feeding into the builder. By "normalize",
91     // we mean that all precomposed Unicode characters must be converted into
92     // a base character and one or more combining characters (such as accents).
93     // When there are multiple combining characters attached to a base character,
94     // the combining characters must be in their canonical order
95     //
96     // sherman/Note:
97     //(1)decmp will be NO_DECOMPOSITION only in ko locale to prevent decompose
98     //hanguul syllables to jamos, so we can actually just call decompose with
99     //normalizer's IGNORE_HANGUL option turned on
100    //
101    //(2)just call the "special version" in NormalizerImpl directly
102    //pattern = Normalizer.decompose(pattern, false, Normalizer.IGNORE_HANGUL, true);
103    //
104    //Normalizer.Mode mode = CollatorUtilities.toNormalizerMode(decmp);
105    //pattern = Normalizer.normalize(pattern, mode, 0, true);
106
107    pattern = NormalizerImpl.canonicalDecomposeWithSingleQuotation(pattern);
108
109    // Build the merged collation entries
110    // Since rules can be specified in any order in the string
111    // (e.g. "c , C < d , D < e , E .... C < CH")
112    // this splits all of the rules in the string out into separate
113    // objects and then sorts them. In the above example, it merges the
114    // "C < CH" rule in just before the "C < D" rule.
115    //
116
117    mPattern = new MergeCollation(pattern);
118
119    int order = 0;
120
121    // Now walk through each entry and add it to my own tables
122    for (i = 0; i < mPattern.getCount(); ++i)
123    {
124        PatternEntry entry = mPattern.getItemAt(i);
125        if (entry != null) {
126            groupChars = entry.getChars();
127            if (groupChars.length() > 1) {
128                switch(groupChars.charAt(groupChars.length()-1)) {
129                    case '@':
130                        frenchSec = true;
131                        groupChars = groupChars.substring(0, groupChars.length()-1);
132                        break;
133                    case '!':
134                        seAsianSwapping = true;
135                        groupChars = groupChars.substring(0, groupChars.length()-1);
136                        break;
137                }
138            }
139
140            order = increment(entry.getStrength(), order);
141            expChars = entry.getExtension();
142

```

```

143         if (expChars.length() != 0) {
144             addExpandOrder(groupChars, expChars, order);
145         } else if (groupChars.length() > 1) {
146             char ch = groupChars.charAt(0);
147             if (Character.isHighSurrogate(ch) && groupChars.length() == 2) {
148                 addOrder(Character.toCodePoint(ch, groupChars.charAt(1)), order);
149             } else {
150                 addContractOrder(groupChars, order);
151             }
152         } else {
153             char ch = groupChars.charAt(0);
154             addOrder(ch, order);
155         }
156     }
157 }
158 addComposedChars();
159
160 commit();
161 mapping.compact();
162 /*
163 System.out.println("mappingSize=" + mapping.getKSize());
164 for (int j = 0; j < 0xffff; j++) {
165     int value = mapping.elementAt(j);
166     if (value != RBCollationTables.UNMAPPED)
167         System.out.println("index=" + Integer.toString(j, 16)
168             + ", value=" + Integer.toString(value, 16));
169 }
170 */
171 tables.fillInTables(frenchSec, seAsianSwapping, mapping, contractTable, expandTable,
172     contractFlags, maxSecOrder, maxTerOrder);
173 }
174
175 /** Add expanding entries for pre-composed unicode characters so that this
176  * collator can be used reasonably well with decomposition turned off.
177  */
178 private void addComposedChars() throws ParseException {
179     // Iterate through all of the pre-composed characters in Unicode
180     ComposedCharIter iter = new ComposedCharIter();
181     int c;
182     while ((c = iter.next()) != ComposedCharIter.DONE) {
183         if (getCharOrder(c) == RBCollationTables.UNMAPPED) {
184             //
185             // We don't already have an ordering for this pre-composed character.
186             //
187             // First, see if the decomposed string is already in our
188             // tables as a single contracting-string ordering.
189             // If so, just map the precomposed character to that order.
190             //
191             // TODO: What we should really be doing here is trying to find the
192             // longest initial substring of the decomposition that is present
193             // in the tables as a contracting character sequence, and find its
194             // ordering. Then do this recursively with the remaining chars
195             // so that we build a list of orderings, and add that list to

```

```

196         // the expansion table.
197         // That would be more correct but also significantly slower, so
198         // I'm not totally sure it's worth doing.
199         //
200         String s = iter.decomposition();
201
202         //sherman/Note: if this is 1 character decomposed string, the
203         //only thing need to do is to check if this decomposed character
204         //has an entry in our order table, this order is not necessary
205         //to be a contraction order, if it does have one, add an entry
206         //for the precomposed character by using the same order, the
207         //previous impl unnecessarily adds a single character expansion
208         //entry.
209         if (s.length() == 1) {
210             int order = getCharOrder(s.charAt(0));
211             if (order != RBCollationTables.UNMAPPED) {
212                 addOrder(c, order);
213             }
214             continue;
215         } else if (s.length() == 2) {
216             char ch0 = s.charAt(0);
217             if (Character.isHighSurrogate(ch0)) {
218                 int order = getCharOrder(s.codePointAt(0));
219                 if (order != RBCollationTables.UNMAPPED) {
220                     addOrder(c, order);
221                 }
222                 continue;
223             }
224         }
225         int contractOrder = getContractOrder(s);
226         if (contractOrder != RBCollationTables.UNMAPPED) {
227             addOrder(c, contractOrder);
228         } else {
229             //
230             // We don't have a contracting ordering for the entire string
231             // that results from the decomposition, but if we have orders
232             // for each individual character, we can add an expanding
233             // table entry for the pre-composed character
234             //
235             boolean allThere = true;
236             for (int i = 0; i < s.length(); i++) {
237                 if (getCharOrder(s.charAt(i)) == RBCollationTables.UNMAPPED) {
238                     allThere = false;
239                     break;
240                 }
241             }
242             if (allThere) {
243                 addExpandOrder(c, s, RBCollationTables.UNMAPPED);
244             }
245         }
246     }
247 }
248 }

```

```

249
250 /**
251  * Look up for unmapped values in the expanded character table.
252  *
253  * When the expanding character tables are built by addExpandOrder,
254  * it doesn't know what the final ordering of each character
255  * in the expansion will be. Instead, it just puts the raw character
256  * code into the table, adding CHARINDEX as a flag. Now that we've
257  * finished building the mapping table, we can go back and look up
258  * that character to see what its real collation order is and
259  * stick that into the expansion table. That lets us avoid doing
260  * a two-stage lookup later.
261  */
262 private final void commit()
263 {
264     if (expandTable != null) {
265         for (int i = 0; i < expandTable.size(); i++) {
266             int[] valueList = expandTable.elementAt(i);
267             for (int j = 0; j < valueList.length; j++) {
268                 int order = valueList[j];
269                 if (order < RBCollationTables.EXPANDCHARINDEX && order > CHARINDEX) {
270                     // found a expanding character that isn't filled in yet
271                     int ch = order - CHARINDEX;
272
273                     // Get the real values for the non-filled entry
274                     int realValue = getCharOrder(ch);
275
276                     if (realValue == RBCollationTables.UNMAPPED) {
277                         // The real value is still unmapped, maybe it's ignorable
278                         valueList[j] = IGNORABLEMASK & ch;
279                     } else {
280                         // just fill in the value
281                         valueList[j] = realValue;
282                     }
283                 }
284             }
285         }
286     }
287 }
288 /**
289  * Increment of the last order based on the comparison level.
290  */
291 private final int increment(int aStrength, int lastValue)
292 {
293     switch(aStrength)
294     {
295     case Collator.PRIMARY:
296         // increment primary order and mask off secondary and tertiary difference
297         lastValue += PRIMARYORDERINCREMENT;
298         lastValue &= RBCollationTables.PRIMARYORDERMASK;
299         isOverIgnore = true;
300         break;
301     case Collator.SECONDARY:

```

```

302         // increment secondary order and mask off tertiary difference
303         lastValue += SECONDARYORDERINCREMENT;
304         lastValue &= RBCollationTables.SECONDARYDIFFERENCEONLY;
305         // record max # of ignorable chars with secondary difference
306         if (!isOverIgnore)
307             maxSecOrder++;
308         break;
309     case Collator.TERTIARY:
310         // increment tertiary order
311         lastValue += TERTIARYORDERINCREMENT;
312         // record max # of ignorable chars with tertiary difference
313         if (!isOverIgnore)
314             maxTerOrder++;
315         break;
316     }
317     return lastValue;
318 }
319
320 /**
321  * Adds a character and its designated order into the collation table.
322  */
323 private final void addOrder(int ch, int anOrder)
324 {
325     // See if the char already has an order in the mapping table
326     int order = mapping.elementAt(ch);
327
328     if (order >= RBCollationTables.CONTRACTCHARINDEX) {
329         // There's already an entry for this character that points to a contracting
330         // character table. Instead of adding the character directly to the mapping
331         // table, we must add it to the contract table instead.
332         int length = 1;
333         if (Character.isSupplementaryCodePoint(ch)) {
334             length = Character.toChars(ch, keyBuf, 0);
335         } else {
336             keyBuf[0] = (char)ch;
337         }
338         addContractOrder(new String(keyBuf, 0, length), anOrder);
339     } else {
340         // add the entry to the mapping table,
341         // the same later entry replaces the previous one
342         mapping.setElementAt(ch, anOrder);
343     }
344 }
345
346 private final void addContractOrder(String groupChars, int anOrder) {
347     addContractOrder(groupChars, anOrder, true);
348 }
349
350 /**
351  * Adds the contracting string into the collation table.
352  */
353 private final void addContractOrder(String groupChars, int anOrder,
354                                     boolean fwd)

```

```

355 {
356     if (contractTable == null) {
357         contractTable = new Vector<>(INITIALTABLESIZE);
358     }
359
360     //initial character
361     int ch = groupChars.codePointAt(0);
362     /*
363     char ch0 = groupChars.charAt(0);
364     int ch = Character.isHighSurrogate(ch0)?
365         Character.toCodePoint(ch0, groupChars.charAt(1)):ch0;
366     */
367     // See if the initial character of the string already has a contract table.
368     int entry = mapping.elementAt(ch);
369     Vector<EntryPair> entryTable = getContractValuesImpl(entry - RBCollationTables.
        CONTRACTCHARINDEX);
370
371     if (entryTable == null) {
372         // We need to create a new table of contract entries for this base char
373         int tableIndex = RBCollationTables.CONTRACTCHARINDEX + contractTable.size();
374         entryTable = new Vector<>(INITIALTABLESIZE);
375         contractTable.addElement(entryTable);
376
377         // Add the initial character's current ordering first. then
378         // update its mapping to point to this contract table
379         entryTable.addElement(new EntryPair(groupChars.substring(0,Character.charCount(ch)),
            entry));
380         mapping.setElementAt(ch, tableIndex);
381     }
382
383     // Now add (or replace) this string in the table
384     int index = RBCollationTables.getEntry(entryTable, groupChars, fwd);
385     if (index != RBCollationTables.UNMAPPED) {
386         EntryPair pair = entryTable.elementAt(index);
387         pair.value = anOrder;
388     } else {
389         EntryPair pair = entryTable.lastElement();
390
391         // NOTE: This little bit of logic is here to speed CollationElementIterator
392         // .nextContractChar(). This code ensures that the longest sequence in
393         // this list is always the _last_ one in the list. This keeps
394         // nextContractChar() from having to search the entire list for the longest
395         // sequence.
396         if (groupChars.length() > pair.entryName.length()) {
397             entryTable.addElement(new EntryPair(groupChars, anOrder, fwd));
398         } else {
399             entryTable.insertElementAt(new EntryPair(groupChars, anOrder,
400                 fwd), entryTable.size() - 1);
401         }
402     }
403
404     // If this was a forward mapping for a contracting string, also add a
405     // reverse mapping for it, so that CollationElementIterator.previous

```



```

406     // can work right
407     if (fwd && groupChars.length() > 1) {
408         addContractFlags(groupChars);
409         addContractOrder(new StringBuffer(groupChars).reverse().toString(),
410             anOrder, false);
411     }
412 }
413
414 /**
415  * If the given string has been specified as a contracting string
416  * in this collation table, return its ordering.
417  * Otherwise return UNMAPPED.
418  */
419 private int getContractOrder(String groupChars)
420 {
421     int result = RBCollationTables.UNMAPPED;
422     if (contractTable != null) {
423         int ch = groupChars.codePointAt(0);
424         /*
425          char ch0 = groupChars.charAt(0);
426          int ch = Character.isHighSurrogate(ch0)?
427              Character.toCodePoint(ch0, groupChars.charAt(1)):ch0;
428          */
429         Vector<EntryPair> entryTable = getContractValues(ch);
430         if (entryTable != null) {
431             int index = RBCollationTables.getEntry(entryTable, groupChars, true);
432             if (index != RBCollationTables.UNMAPPED) {
433                 EntryPair pair = entryTable.elementAt(index);
434                 result = pair.value;
435             }
436         }
437     }
438     return result;
439 }
440
441 private final int getCharOrder(int ch) {
442     int order = mapping.elementAt(ch);
443
444     if (order >= RBCollationTables.CONTRACTCHARINDEX) {
445         Vector<EntryPair> groupList = getContractValuesImpl(order - RBCollationTables.
446             CONTRACTCHARINDEX);
447         EntryPair pair = groupList.firstElement();
448         order = pair.value;
449     }
450     return order;
451 }
452
453 /**
454  * Get the entry of hash table of the contracting string in the collation
455  * table.
456  * @param ch the starting character of the contracting string
457  */
458 private Vector<EntryPair> getContractValues(int ch)

```

```

458     {
459         int index = mapping.elementAt(ch);
460         return getContractValuesImpl(index - RBCollationTables.CONTRACTCHARINDEX);
461     }
462
463     private Vector<EntryPair> getContractValuesImpl(int index)
464     {
465         if (index >= 0)
466         {
467             return contractTable.elementAt(index);
468         }
469         else // not found
470         {
471             return null;
472         }
473     }
474
475     /**
476     * Adds the expanding string into the collation table.
477     */
478     private final void addExpandOrder(String contractChars,
479                                     String expandChars,
480                                     int anOrder) throws ParseException
481     {
482         // Create an expansion table entry
483         int tableIndex = addExpansion(anOrder, expandChars);
484
485         // And add its index into the main mapping table
486         if (contractChars.length() > 1) {
487             char ch = contractChars.charAt(0);
488             if (Character.isHighSurrogate(ch) && contractChars.length() == 2) {
489                 char ch2 = contractChars.charAt(1);
490                 if (Character.isLowSurrogate(ch2)) {
491                     //only add into table when it is a legal surrogate
492                     addOrder(Character.toCodePoint(ch, ch2), tableIndex);
493                 }
494             } else {
495                 addContractOrder(contractChars, tableIndex);
496             }
497         } else {
498             addOrder(contractChars.charAt(0), tableIndex);
499         }
500     }
501
502     private final void addExpandOrder(int ch, String expandChars, int anOrder)
503     throws ParseException
504     {
505         int tableIndex = addExpansion(anOrder, expandChars);
506         addOrder(ch, tableIndex);
507     }
508
509     /**
510     * Create a new entry in the expansion table that contains the orderings

```

```

511     * for the given characers.  If anOrder is valid, it is added to the
512     * beginning of the expanded list of orders.
513     */
514     private int addExpansion(int anOrder, String expandChars) {
515         if (expandTable == null) {
516             expandTable = new Vector<>(INITIALTABLESIZE);
517         }
518
519         // If anOrder is valid, we want to add it at the beginning of the list
520         int offset = (anOrder == RBCollationTables.UNMAPPED) ? 0 : 1;
521
522         int[] valueList = new int[expandChars.length() + offset];
523         if (offset == 1) {
524             valueList[0] = anOrder;
525         }
526
527         int j = offset;
528         for (int i = 0; i < expandChars.length(); i++) {
529             char ch0 = expandChars.charAt(i);
530             char ch1;
531             int ch;
532             if (Character.isHighSurrogate(ch0)) {
533                 if (++i == expandChars.length() ||
534                     !Character.isLowSurrogate(ch1=expandChars.charAt(i))) {
535                     //either we are missing the low surrogate or the next char
536                     //is not a legal low surrogate, so stop loop
537                     break;
538                 }
539                 ch = Character.toCodePoint(ch0, ch1);
540
541             } else {
542                 ch = ch0;
543             }
544
545             int mapValue = getCharOrder(ch);
546
547             if (mapValue != RBCollationTables.UNMAPPED) {
548                 valueList[j++] = mapValue;
549             } else {
550                 // can't find it in the table, will be filled in by commit().
551                 valueList[j++] = CHARINDEX + ch;
552             }
553         }
554         if (j < valueList.length) {
555             //we had at least one supplementary character, the size of valueList
556             //is bigger than it really needs...
557             int[] tmpBuf = new int[j];
558             while (--j >= 0) {
559                 tmpBuf[j] = valueList[j];
560             }
561             valueList = tmpBuf;
562         }
563         // Add the expanding char list into the expansion table.

```

```

564         int tableIndex = RBCollationTables.EXPANDCHARINDEX + expandTable.size();
565         expandTable.addElement(valueList);
566
567         return tableIndex;
568     }
569
570     private void addContractFlags(String chars) {
571         char c0;
572         int c;
573         int len = chars.length();
574         for (int i = 0; i < len; i++) {
575             c0 = chars.charAt(i);
576             c = Character.isHighSurrogate(c0)
577                 ? Character.toCodePoint(c0, chars.charAt(++i))
578                 : c0;
579             contractFlags.put(c, 1);
580         }
581     }
582
583     // =====
584     // constants
585     // =====
586     final static int CHARINDEX = 0x70000000; // need look up in .commit()
587
588     private final static int IGNORABLEMASK = 0x0000ffff;
589     private final static int PRIMARYORDERINCREMENT = 0x00010000;
590     private final static int SECONDARYORDERINCREMENT = 0x00000100;
591     private final static int TERTIARYORDERINCREMENT = 0x00000001;
592     private final static int INITIALENTABLESIZE = 20;
593     private final static int MAXKEYSIZE = 5;
594
595     // =====
596     // instance variables
597     // =====
598
599     // variables used by the build process
600     private RBCollationTables.BuildAPI tables = null;
601     private MergeCollation mPattern = null;
602     private boolean isOverIgnore = false;
603     private char[] keyBuf = new char[MAXKEYSIZE];
604     private IntHashtable contractFlags = new IntHashtable(100);
605
606     // "shadow" copies of the instance variables in RBCollationTables
607     // (the values in these variables are copied back into RBCollationTables
608     // at the end of the build process)
609     private boolean frenchSec = false;
610     private boolean seAsianSwapping = false;
611
612     private UCompactIntArray mapping = null;
613     private Vector<Vector<EntryPair>> contractTable = null;
614     private Vector<int[]> expandTable = null;
615
616     private short maxSecOrder = 0;

```

```

617     private short maxTerOrder = 0;
618 }

```

2.31 RuleBasedCollationKey.java

Secuencia 40: java/text/RuleBasedCollationKey.java

```

1  /*
2  * Copyright (c) 2005, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright Taligent, Inc. 1996 - All Rights Reserved
28 * (C) Copyright IBM Corp. 1996 - All Rights Reserved
29 *
30 * The original version of this source code and documentation is copyrighted
31 * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32 * materials are provided under terms of a License Agreement between Taligent
33 * and Sun. This technology is protected by multiple US and International
34 * patents. This notice and attribution to Taligent may not be removed.
35 * Taligent is a registered trademark of Taligent, Inc.
36 *
37 */
38
39 package java.text;
40
41 /**
42 * A RuleBasedCollationKey is a concrete implementation of CollationKey class.
43 * The RuleBasedCollationKey class is used by the RuleBasedCollator class.
44 */
45
46 final class RuleBasedCollationKey extends CollationKey {
47     /**

```

```

48      * Compare this RuleBasedCollationKey to target. The collation rules of the
49      * Collator object which created these keys are applied. <strong>Note:</strong>
50      * RuleBasedCollationKeys created by different Collators can not be compared.
51      * @param target target RuleBasedCollationKey
52      * @return Returns an integer value. Value is less than zero if this is less
53      * than target, value is zero if this and target are equal and value is greater than
54      * zero if this is greater than target.
55      * @see java.text.Collator#compare
56      */
57      public int compareTo(CollationKey target)
58      {
59          int result = key.compareTo(((RuleBasedCollationKey)(target)).key);
60          if (result <= Collator.LESS)
61              return Collator.LESS;
62          else if (result >= Collator.GREATER)
63              return Collator.GREATER;
64          return Collator.EQUAL;
65      }
66
67      /**
68      * Compare this RuleBasedCollationKey and the target for equality.
69      * The collation rules of the Collator object which created these keys are applied.
70      * <strong>Note:</strong> RuleBasedCollationKeys created by different Collators can not be
71      * compared.
72      * @param target the RuleBasedCollationKey to compare to.
73      * @return Returns true if two objects are equal, false otherwise.
74      */
75      public boolean equals(Object target) {
76          if (this == target) return true;
77          if (target == null || !getClass().equals(target.getClass())) {
78              return false;
79          }
80          RuleBasedCollationKey other = (RuleBasedCollationKey)target;
81          return key.equals(other.key);
82      }
83
84      /**
85      * Creates a hash code for this RuleBasedCollationKey. The hash value is calculated on the
86      * key itself, not the String from which the key was created. Thus
87      * if x and y are RuleBasedCollationKeys, then x.hashCode(x) == y.hashCode() if
88      * x.equals(y) is true. This allows language-sensitive comparison in a hash table.
89      * See the CollatinKey class description for an example.
90      * @return the hash value based on the string's collation order.
91      */
92      public int hashCode() {
93          return (key.hashCode());
94      }
95
96      /**
97      * Converts the RuleBasedCollationKey to a sequence of bits. If two RuleBasedCollationKeys
98      * could be legitimately compared, then one could compare the byte arrays
99      * for each of those keys to obtain the same result. Byte arrays are
100     * organized most significant byte first.

```

```

101  */
102  public byte[] toByteArray() {
103
104      char[] src = key.toCharArray();
105      byte[] dest = new byte[ 2*src.length ];
106      int j = 0;
107      for( int i=0; i<src.length; i++ ) {
108          dest[j++] = (byte)(src[i] >>> 8);
109          dest[j++] = (byte)(src[i] & 0x00ff);
110      }
111      return dest;
112  }
113
114  /**
115   * A RuleBasedCollationKey can only be generated by Collator objects.
116   */
117  RuleBasedCollationKey(String source, String key) {
118      super(source);
119      this.key = key;
120  }
121  private String key = null;
122
123  }

```

2.32 RuleBasedCollator.java

Secuencia 41: java/text/RuleBasedCollator.java

```

1  /*
2  * Copyright (c) 1997, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*

```

```

27 * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28 * (C) Copyright IBM Corp. 1996-1998 - All Rights Reserved
29 *
30 * The original version of this source code and documentation is copyrighted
31 * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32 * materials are provided under terms of a License Agreement between Taligent
33 * and Sun. This technology is protected by multiple US and International
34 * patents. This notice and attribution to Taligent may not be removed.
35 * Taligent is a registered trademark of Taligent, Inc.
36 *
37 */
38
39 package java.text;
40
41 import java.text.Normalizer;
42 import java.util.Vector;
43 import java.util.Locale;
44
45 /**
46 * The RuleBasedCollator class is a concrete subclass of
47 * Collator that provides a simple, data-driven, table
48 * collator. With this class you can create a customized table-based
49 * Collator. RuleBasedCollator maps
50 * characters to sort keys.
51 *
52 * <p>
53 * RuleBasedCollator has the following restrictions
54 * for efficiency (other subclasses may be used for more complex languages) :
55 * <ol>
56 * <li>If a special collation rule controlled by a <modifier> is
57 * specified it applies to the whole collator object.
58 * <li>All non-mentioned characters are at the end of the
59 * collation order.
60 * </ol>
61 *
62 * <p>
63 * The collation table is composed of a list of collation rules, where each
64 * rule is of one of three forms:
65 * <pre>
66 * <modifier>;
67 * <relation>; <text-argument>;
68 * <reset>; <text-argument>;
69 * </pre>
70 * The definitions of the rule elements is as follows:
71 * <UL>
72 * <LI><strong>Text-Argument</strong>: A text-argument is any sequence of
73 * characters, excluding special characters (that is, common
74 * whitespace characters [0009-000D, 0020] and rule syntax characters
75 * [0021-002F, 003A-0040, 005B-0060, 007B-007E]). If those
76 * characters are desired, you can put them in single quotes
77 * (e.g. ampersand =<code>'&&'</code>). Note that unquoted white space characters
78 * are ignored; e.g. <code>b c</code> is treated as <code>bc</code>.
79 * <LI><strong>Modifier</strong>: There are currently two modifiers that

```



```

80 *      turn on special collation rules.
81 *      <UL>
82 *          <LI>'@' : Turns on backwards sorting of accents (secondary
83 *                  differences), as in French.
84 *          <LI>'!' : Turns on Thai/Lao vowel-consonant swapping. If this
85 *                  rule is in force when a Thai vowel of the range
86 *                  &#92;U0E40-&#92;U0E44 precedes a Thai consonant of the range
87 *                  &#92;U0E01-&#92;U0E2E OR a Lao vowel of the range &#92;U0EC0-&#92;U0EC4
88 *                  precedes a Lao consonant of the range &#92;U0E81-&#92;U0EAE then
89 *                  the vowel is placed after the consonant for collation
90 *                  purposes.
91 *      </UL>
92 *      <p>'@' : Indicates that accents are sorted backwards, as in French.
93 *      <LI><strong>Relation</strong>: The relations are the following:
94 *      <UL>
95 *          <LI>'&lt;' : Greater, as a letter difference (primary)
96 *          <LI>';' : Greater, as an accent difference (secondary)
97 *          <LI>', ' : Greater, as a case difference (tertiary)
98 *          <LI>'=' : Equal
99 *      </UL>
100 *      <LI><strong>Reset</strong>: There is a single reset
101 *          which is used primarily for contractions and expansions, but which
102 *          can also be used to add a modification at the end of a set of rules.
103 *      <p>'&' : Indicates that the next rule follows the position to where
104 *          the reset text-argument would be sorted.
105 * </UL>
106 *
107 * <p>
108 * This sounds more complicated than it is in practice. For example, the
109 * following are equivalent ways of expressing the same thing:
110 * <blockquote>
111 * <pre>
112 * a &lt; b &lt; c
113 * a &lt; b & amp; b &lt; c
114 * a &lt; c & amp; a &lt; b
115 * </pre>
116 * </blockquote>
117 * Notice that the order is important, as the subsequent item goes immediately
118 * after the text-argument. The following are not equivalent:
119 * <blockquote>
120 * <pre>
121 * a &lt; b & amp; a &lt; c
122 * a &lt; c & amp; a &lt; b
123 * </pre>
124 * </blockquote>
125 * Either the text-argument must already be present in the sequence, or some
126 * initial substring of the text-argument must be present. (e.g. "a &lt; b & amp; ae &lt;
127 * e" is valid since "a" is present in the sequence before "ae" is reset). In
128 * this latter case, "ae" is not entered and treated as a single character;
129 * instead, "e" is sorted as if it were expanded to two characters: "a"
130 * followed by an "e". This difference appears in natural languages: in
131 * traditional Spanish "ch" is treated as though it contracts to a single
132 * character (expressed as "c &lt; ch &lt; d"), while in traditional German

```

```

133 * a-umlaut is treated as though it expanded to two characters
134 * (expressed as "a,A &lt; b,B ... &#92;u00e3&#92;u00c3").
135 * [&#92;u00e3 and &#92;u00c3 are, of course, the escape sequences for a-umlaut.]
136 * <p>
137 * <strong>Ignorable Characters</strong>
138 * <p>
139 * For ignorable characters, the first rule must start with a relation (the
140 * examples we have used above are really fragments; "a &lt; b" really should be
141 * "&lt; a &lt; b"). If, however, the first relation is not "&lt;", then all the all
142 * text-arguments up to the first "&lt;" are ignorable. For example, ", - &lt; a &lt; b"
143 * makes "-" an ignorable character, as we saw earlier in the word
144 * "black-birds". In the samples for different languages, you see that most
145 * accents are ignorable.
146 *
147 * <p><strong>Normalization and Accents</strong>
148 * <p>
149 * <code>RuleBasedCollator</code> automatically processes its rule table to
150 * include both pre-composed and combining-character versions of
151 * accented characters. Even if the provided rule string contains only
152 * base characters and separate combining accent characters, the pre-composed
153 * accented characters matching all canonical combinations of characters from
154 * the rule string will be entered in the table.
155 * <p>
156 * This allows you to use a RuleBasedCollator to compare accented strings
157 * even when the collator is set to NO_DECOMPOSITION. There are two caveats,
158 * however. First, if the strings to be collated contain combining
159 * sequences that may not be in canonical order, you should set the collator to
160 * CANONICAL_DECOMPOSITION or FULL_DECOMPOSITION to enable sorting of
161 * combining sequences. Second, if the strings contain characters with
162 * compatibility decompositions (such as full-width and half-width forms),
163 * you must use FULL_DECOMPOSITION, since the rule tables only include
164 * canonical mappings.
165 *
166 * <p><strong>Errors</strong>
167 * <p>
168 * The following are errors:
169 * <UL>
170 * <LI>A text-argument contains unquoted punctuation symbols
171 * (e.g. "a &lt; b-c &lt; d").
172 * <LI>A relation or reset character not followed by a text-argument
173 * (e.g. "a &lt; ,b").
174 * <LI>A reset where the text-argument (or an initial substring of the
175 * text-argument) is not already in the sequence.
176 * (e.g. "a &lt; b &#92; e &lt; f")
177 * </UL>
178 * If you produce one of these errors, a <code>RuleBasedCollator</code> throws
179 * a <code>ParseException</code>.
180 *
181 * <p><strong>Examples</strong>
182 * <p>Simple: "&lt; a &lt; b &lt; c &lt; d"
183 * <p>Norwegian: "&lt; a, A &lt; b, B &lt; c, C &lt; d, D &lt; e, E &lt; f, F
184 * &lt; g, G &lt; h, H &lt; i, I &lt; j, J &lt; k, K &lt; l, L
185 * &lt; m, M &lt; n, N &lt; o, O &lt; p, P &lt; q, Q &lt; r, R

```

```

186 *      &lt; s, S &lt; t, T &lt; u, U &lt; v, V &lt; w, W &lt; x, X
187 *      &lt; y, Y &lt; z, Z
188 *      &lt; &#92;u00E6, &#92;u00C6
189 *      &lt; &#92;u00F8, &#92;u00D8
190 *      &lt; &#92;u00E5 = a&#92;u030A, &#92;u00C5 = A&#92;u030A;
191 *      aa, AA"
192 *
193 * <p>
194 * To create a <code>RuleBasedCollator</code> object with specialized
195 * rules tailored to your needs, you construct the <code>RuleBasedCollator</code>
196 * with the rules contained in a <code>String</code> object. For example:
197 * <blockquote>
198 * <pre>
199 * String simple = "&lt; a&lt; b&lt; c&lt; d";
200 * RuleBasedCollator mySimple = new RuleBasedCollator(simple);
201 * </pre>
202 * </blockquote>
203 * Or:
204 * <blockquote>
205 * <pre>
206 * String Norwegian = "&lt; a, A &lt; b, B &lt; c, C &lt; d, D &lt; e, E &lt; f, F &lt; g, G &lt; h
    , H &lt; i, I" +
207 *      "&lt; j, J &lt; k, K &lt; l, L &lt; m, M &lt; n, N &lt; o, O &lt; p, P &lt; q
    , Q &lt; r, R" +
208 *      "&lt; s, S &lt; t, T &lt; u, U &lt; v, V &lt; w, W &lt; x, X &lt; y, Y &lt; z
    , Z" +
209 *      "&lt; &#92;u00E6, &#92;u00C6" +      // Latin letter ae &amp; AE
210 *      "&lt; &#92;u00F8, &#92;u00D8" +      // Latin letter o &amp; O with stroke
211 *      "&lt; &#92;u00E5 = a&#92;u030A," +    // Latin letter a with ring above
212 *      " &#92;u00C5 = A&#92;u030A;" +    // Latin letter A with ring above
213 *      " aa, AA";
214 * RuleBasedCollator myNorwegian = new RuleBasedCollator(Norwegian);
215 * </pre>
216 * </blockquote>
217 *
218 * <p>
219 * A new collation rules string can be created by concatenating rules
220 * strings. For example, the rules returned by {@link #getRules()} could
221 * be concatenated to combine multiple <code>RuleBasedCollator</code>s.
222 *
223 * <p>
224 * The following example demonstrates how to change the order of
225 * non-spacing accents,
226 * <blockquote>
227 * <pre>
228 * // old rule
229 * String oldRules = "&#92;u0301;&#92;u0300;&#92;u0302;&#92;u0308"    // main accents
230 *      + "&#92;u0327;&#92;u0303;&#92;u0304;&#92;u0305"    // main accents
231 *      + "&#92;u0306;&#92;u0307;&#92;u0309;&#92;u030A"    // main accents
232 *      + "&#92;u030B;&#92;u030C;&#92;u030D;&#92;u030E"    // main accents
233 *      + "&#92;u030F;&#92;u0310;&#92;u0311;&#92;u0312"    // main accents
234 *      + "&lt; a , A ; ae, AE ; &#92;u00e6 , &#92;u00c6"
235 *      + "&lt; b , B &lt; c, C &lt; e, E &amp; C &lt; d, D";

```

```

236 * // change the order of accent characters
237 * String addOn = "&#92;u0300 ; &#92;u0308 ; &#92;u0302";
238 * RuleBasedCollator myCollator = new RuleBasedCollator(oldRules + addOn);
239 * </pre>
240 * </blockquote>
241 *
242 * @see      Collator
243 * @see      CollationElementIterator
244 * @author    Helena Shih, Laura Werner, Richard Gillam
245 */
246 public class RuleBasedCollator extends Collator{
247     // IMPLEMENTATION NOTES: The implementation of the collation algorithm is
248     // divided across three classes: RuleBasedCollator, RBCollationTables, and
249     // CollationElementIterator. RuleBasedCollator contains the collator's
250     // transient state and includes the code that uses the other classes to
251     // implement comparison and sort-key building. RuleBasedCollator also
252     // contains the logic to handle French secondary accent sorting.
253     // A RuleBasedCollator has two CollationElementIterators. State doesn't
254     // need to be preserved in these objects between calls to compare() or
255     // getCollationKey(), but the objects persist anyway to avoid wasting extra
256     // creation time. compare() and getCollationKey() are synchronized to ensure
257     // thread safety with this scheme. The CollationElementIterator is responsible
258     // for generating collation elements from strings and returning one element at
259     // a time (sometimes there's a one-to-many or many-to-one mapping between
260     // characters and collation elements-- this class handles that).
261     // CollationElementIterator depends on RBCollationTables, which contains the
262     // collator's static state. RBCollationTables contains the actual data
263     // tables specifying the collation order of characters for a particular locale
264     // or use. It also contains the base logic that CollationElementIterator
265     // uses to map from characters to collation elements. A single RBCollationTables
266     // object is shared among all RuleBasedCollators for the same locale, and
267     // thus by all the CollationElementIterators they create.
268
269     /**
270      * RuleBasedCollator constructor. This takes the table rules and builds
271      * a collation table out of them. Please see RuleBasedCollator class
272      * description for more details on the collation rule syntax.
273      * @see java.util.Locale
274      * @param rules the collation rules to build the collation table from.
275      * @exception ParseException A format exception
276      * will be thrown if the build process of the rules fails. For
277      * example, build rule "a &lt; ? &lt; d" will cause the constructor to
278      * throw the ParseException because the '?' is not quoted.
279      */
280     public RuleBasedCollator(String rules) throws ParseException {
281         this(rules, Collator.CANONICAL_DECOMPOSITION);
282     }
283
284     /**
285      * RuleBasedCollator constructor. This takes the table rules and builds
286      * a collation table out of them. Please see RuleBasedCollator class
287      * description for more details on the collation rule syntax.
288      * @see java.util.Locale

```

```

289     * @param rules the collation rules to build the collation table from.
290     * @param decomp the decomposition strength used to build the
291     * collation table and to perform comparisons.
292     * @exception ParseException A format exception
293     * will be thrown if the build process of the rules fails. For
294     * example, build rule "a < ? < d" will cause the constructor to
295     * throw the ParseException because the '?' is not quoted.
296     */
297     RuleBasedCollator(String rules, int decomp) throws ParseException {
298         setStrength(Collator.TERTIARY);
299         setDecomposition(decomp);
300         tables = new RBCollationTables(rules, decomp);
301     }
302
303     /**
304     * "Copy constructor." Used in clone() for performance.
305     */
306     private RuleBasedCollator(RuleBasedCollator that) {
307         setStrength(that.getStrength());
308         setDecomposition(that.getDecomposition());
309         tables = that.tables;
310     }
311
312     /**
313     * Gets the table-based rules for the collation object.
314     * @return returns the collation rules that the table collation object
315     * was created from.
316     */
317     public String getRules()
318     {
319         return tables.getRules();
320     }
321
322     /**
323     * Returns a CollationElementIterator for the given String.
324     *
325     * @param source the string to be collated
326     * @return a {@code CollationElementIterator} object
327     * @see java.text.CollationElementIterator
328     */
329     public CollationElementIterator getCollationElementIterator(String source) {
330         return new CollationElementIterator( source, this );
331     }
332
333     /**
334     * Returns a CollationElementIterator for the given CharacterIterator.
335     *
336     * @param source the character iterator to be collated
337     * @return a {@code CollationElementIterator} object
338     * @see java.text.CollationElementIterator
339     * @since 1.2
340     */
341     public CollationElementIterator getCollationElementIterator(

```

```

342         CharacterIterator source) {
343     return new CollationElementIterator( source, this );
344 }
345
346 /**
347  * Compares the character data stored in two different strings based on the
348  * collation rules. Returns information about whether a string is less
349  * than, greater than or equal to another string in a language.
350  * This can be overridden in a subclass.
351  *
352  * @exception NullPointerException if <code>source</code> or <code>target</code> is null.
353  */
354 public synchronized int compare(String source, String target)
355 {
356     if (source == null || target == null) {
357         throw new NullPointerException();
358     }
359
360     // The basic algorithm here is that we use CollationElementIterators
361     // to step through both the source and target strings. We compare each
362     // collation element in the source string against the corresponding one
363     // in the target, checking for differences.
364     //
365     // If a difference is found, we set <result> to LESS or GREATER to
366     // indicate whether the source string is less or greater than the target.
367     //
368     // However, it's not that simple. If we find a tertiary difference
369     // (e.g. 'A' vs. 'a') near the beginning of a string, it can be
370     // overridden by a primary difference (e.g. "A" vs. "B") later in
371     // the string. For example, "AA" < "aB", even though 'A' > 'a'.
372     //
373     // To keep track of this, we use strengthResult to keep track of the
374     // strength of the most significant difference that has been found
375     // so far. When we find a difference whose strength is greater than
376     // strengthResult, it overrides the last difference (if any) that
377     // was found.
378
379     int result = Collator.EQUAL;
380
381     if (sourceCursor == null) {
382         sourceCursor = getCollationElementIterator(source);
383     } else {
384         sourceCursor.setText(source);
385     }
386     if (targetCursor == null) {
387         targetCursor = getCollationElementIterator(target);
388     } else {
389         targetCursor.setText(target);
390     }
391
392     int sOrder = 0, tOrder = 0;
393
394     boolean initialCheckSecTer = getStrength() >= Collator.SECONDARY;

```

```

395     boolean checkSecTer = initialCheckSecTer;
396     boolean checkTertiary = getStrength() >= Collator.TERTIARY;
397
398     boolean gets = true, gett = true;
399
400     while(true) {
401         // Get the next collation element in each of the strings, unless
402         // we've been requested to skip it.
403         if (gets) sOrder = sourceCursor.next(); else gets = true;
404         if (gett) tOrder = targetCursor.next(); else gett = true;
405
406         // If we've hit the end of one of the strings, jump out of the loop
407         if ((sOrder == CollationElementIterator.NULLORDER) ||
408             (tOrder == CollationElementIterator.NULLORDER))
409             break;
410
411         int pSOrder = CollationElementIterator.primaryOrder(sOrder);
412         int pTOrder = CollationElementIterator.primaryOrder(tOrder);
413
414         // If there's no difference at this position, we can skip it
415         if (sOrder == tOrder) {
416             if (tables.isFrenchSec() && pSOrder != 0) {
417                 if (!checkSecTer) {
418                     // in french, a secondary difference more to the right is stronger,
419                     // so accents have to be checked with each base element
420                     checkSecTer = initialCheckSecTer;
421                     // but tertiary differences are less important than the first
422                     // secondary difference, so checking tertiary remains disabled
423                     checkTertiary = false;
424                 }
425             }
426             continue;
427         }
428
429         // Compare primary differences first.
430         if ( pSOrder != pTOrder )
431         {
432             if (sOrder == 0) {
433                 // The entire source element is ignorable.
434                 // Skip to the next source element, but don't fetch another target element.
435                 gett = false;
436                 continue;
437             }
438             if (tOrder == 0) {
439                 gets = false;
440                 continue;
441             }
442
443             // The source and target elements aren't ignorable, but it's still possible
444             // for the primary component of one of the elements to be ignorable....
445
446             if (pSOrder == 0) // primary order in source is ignorable
447             {

```

```

448         // The source's primary is ignorable, but the target's isn't. We treat
449         // ignorables
450         // as a secondary difference, so remember that we found one.
451         if (checkSecTer) {
452             result = Collator.GREATER; // (strength is SECONDARY)
453             checkSecTer = false;
454         }
455         // Skip to the next source element, but don't fetch another target element.
456         gett = false;
457     }
458     else if (pTOrder == 0)
459     {
460         // record differences - see the comment above.
461         if (checkSecTer) {
462             result = Collator.LESS; // (strength is SECONDARY)
463             checkSecTer = false;
464         }
465         // Skip to the next source element, but don't fetch another target element.
466         gets = false;
467     } else {
468         // Neither of the orders is ignorable, and we already know that the primary
469         // orders are different because of the (pSOrder != pTOrder) test above.
470         // Record the difference and stop the comparison.
471         if (pSOrder < pTOrder) {
472             return Collator.LESS; // (strength is PRIMARY)
473         } else {
474             return Collator.GREATER; // (strength is PRIMARY)
475         }
476     }
477 } else { // else of if ( pSOrder != pTOrder )
478     // primary order is the same, but complete order is different. So there
479     // are no base elements at this point, only ignorables (Since the strings are
480     // normalized)
481
482     if (checkSecTer) {
483         // a secondary or tertiary difference may still matter
484         short secSOrder = CollationElementIterator.secondaryOrder(sOrder);
485         short secTOrder = CollationElementIterator.secondaryOrder(tOrder);
486         if (secSOrder != secTOrder) {
487             // there is a secondary difference
488             result = (secSOrder < secTOrder) ? Collator.LESS : Collator.GREATER;
489             // (strength is SECONDARY)
490             checkSecTer = false;
491             // (even in french, only the first secondary difference within
492             // a base character matters)
493         } else {
494             if (checkTertiary) {
495                 // a tertiary difference may still matter
496                 short terSOrder = CollationElementIterator.tertiaryOrder(sOrder);
497                 short terTOrder = CollationElementIterator.tertiaryOrder(tOrder);
498                 if (terSOrder != terTOrder) {
499                     // there is a tertiary difference
500                     result = (terSOrder < terTOrder) ? Collator.LESS : Collator.GREATER

```



```

;
500                                     // (strength is TERTIARY)
501         checkTertiary = false;
502     }
503 }
504 }
505 } // if (checkSecTer)
506
507 } // if ( pSOrder != pTOrder )
508 } // while()
509
510 if (sOrder != CollationElementIterator.NULLORDER) {
511     // (tOrder must be CollationElementIterator::NULLORDER,
512     // since this point is only reached when sOrder or tOrder is NULLORDER.)
513     // The source string has more elements, but the target string hasn't.
514     do {
515         if (CollationElementIterator.primaryOrder(sOrder) != 0) {
516             // We found an additional non-ignorable base character in the source string.
517             // This is a primary difference, so the source is greater
518             return Collator.GREATER; // (strength is PRIMARY)
519         }
520         else if (CollationElementIterator.secondaryOrder(sOrder) != 0) {
521             // Additional secondary elements mean the source string is greater
522             if (checkSecTer) {
523                 result = Collator.GREATER; // (strength is SECONDARY)
524                 checkSecTer = false;
525             }
526         }
527     } while ((sOrder = sourceCursor.next()) != CollationElementIterator.NULLORDER);
528 }
529 else if (tOrder != CollationElementIterator.NULLORDER) {
530     // The target string has more elements, but the source string hasn't.
531     do {
532         if (CollationElementIterator.primaryOrder(tOrder) != 0)
533             // We found an additional non-ignorable base character in the target string.
534             // This is a primary difference, so the source is less
535             return Collator.LESS; // (strength is PRIMARY)
536         else if (CollationElementIterator.secondaryOrder(tOrder) != 0) {
537             // Additional secondary elements in the target mean the source string is less
538             if (checkSecTer) {
539                 result = Collator.LESS; // (strength is SECONDARY)
540                 checkSecTer = false;
541             }
542         }
543     } while ((tOrder = targetCursor.next()) != CollationElementIterator.NULLORDER);
544 }
545
546 // For IDENTICAL comparisons, we use a bitwise character comparison
547 // as a tiebreaker if all else is equal
548 if (result == 0 && getStrength() == IDENTICAL) {
549     int mode = getDecomposition();
550     Normalizer.Form form;
551     if (mode == CANONICAL_DECOMPOSITION) {

```

```

552         form = Normalizer.Form.NFD;
553     } else if (mode == FULL_DECOMPOSITION) {
554         form = Normalizer.Form.NFKD;
555     } else {
556         return source.compareTo(target);
557     }
558
559     String sourceDecomposition = Normalizer.normalize(source, form);
560     String targetDecomposition = Normalizer.normalize(target, form);
561     return sourceDecomposition.compareTo(targetDecomposition);
562 }
563 return result;
564 }
565
566 /**
567  * Transforms the string into a series of characters that can be compared
568  * with CollationKey.compareTo. This overrides java.text.Collator.getCollationKey.
569  * It can be overridden in a subclass.
570  */
571 public synchronized CollationKey getCollationKey(String source)
572 {
573     //
574     // The basic algorithm here is to find all of the collation elements for each
575     // character in the source string, convert them to a char representation,
576     // and put them into the collation key. But it's trickier than that.
577     // Each collation element in a string has three components: primary (A vs B),
578     // secondary (A vs A-acute), and tertiary (A' vs a); and a primary difference
579     // at the end of a string takes precedence over a secondary or tertiary
580     // difference earlier in the string.
581     //
582     // To account for this, we put all of the primary orders at the beginning of the
583     // string, followed by the secondary and tertiary orders, separated by nulls.
584     //
585     // Here's a hypothetical example, with the collation element represented as
586     // a three-digit number, one digit for primary, one for secondary, etc.
587     //
588     // String:           A      a      B   \u00e9 <--(e-acute)
589     // Collation Elements: 101   100   201  510
590     //
591     // Collation Key:     1125<null>0001<null>1010
592     //
593     // To make things even trickier, secondary differences (accent marks) are compared
594     // starting at the *end* of the string in languages with French secondary ordering.
595     // But when comparing the accent marks on a single base character, they are compared
596     // from the beginning. To handle this, we reverse all of the accents that belong
597     // to each base character, then we reverse the entire string of secondary orderings
598     // at the end. Taking the same example above, a French collator might return
599     // this instead:
600     //
601     // Collation Key:     1125<null>1000<null>1010
602     //
603     if (source == null)
604         return null;

```

```

605
606     if (primResult == null) {
607         primResult = new StringBuffer();
608         secResult = new StringBuffer();
609         terResult = new StringBuffer();
610     } else {
611         primResult.setLength(0);
612         secResult.setLength(0);
613         terResult.setLength(0);
614     }
615     int order = 0;
616     boolean compareSec = (getStrength() >= Collator.SECONDARY);
617     boolean compareTer = (getStrength() >= Collator.TERTIARY);
618     int secOrder = CollationElementIterator.NULLORDER;
619     int terOrder = CollationElementIterator.NULLORDER;
620     int preSecIgnore = 0;
621
622     if (sourceCursor == null) {
623         sourceCursor = getCollationElementIterator(source);
624     } else {
625         sourceCursor.setText(source);
626     }
627
628     // walk through each character
629     while ((order = sourceCursor.next()) !=
630         CollationElementIterator.NULLORDER)
631     {
632         secOrder = CollationElementIterator.secondaryOrder(order);
633         terOrder = CollationElementIterator.tertiaryOrder(order);
634         if (!CollationElementIterator.isIgnorable(order))
635         {
636             primResult.append((char) (CollationElementIterator.primaryOrder(order)
637                 + COLLATIONKEYOFFSET));
638
639             if (compareSec) {
640                 //
641                 // accumulate all of the ignorable/secondary characters attached
642                 // to a given base character
643                 //
644                 if (tables.isFrenchSec() && preSecIgnore < secResult.length()) {
645                     //
646                     // We're doing reversed secondary ordering and we've hit a base
647                     // (non-ignorable) character. Reverse any secondary orderings
648                     // that applied to the last base character. (see block comment above.)
649                     //
650                     RBCollationTables.reverse(secResult, preSecIgnore, secResult.length());
651                 }
652                 // Remember where we are in the secondary orderings - this is how far
653                 // back to go if we need to reverse them later.
654                 secResult.append((char)(secOrder+ COLLATIONKEYOFFSET));
655                 preSecIgnore = secResult.length();
656             }
657             if (compareTer) {

```

```

658         terResult.append((char)(terOrder+ COLLATIONKEYOFFSET));
659     }
660 }
661 else
662 {
663     if (compareSec && secOrder != 0)
664         secResult.append((char)
665             (secOrder + tables.getMaxSecOrder() + COLLATIONKEYOFFSET));
666     if (compareTer && terOrder != 0)
667         terResult.append((char)
668             (terOrder + tables.getMaxTerOrder() + COLLATIONKEYOFFSET));
669 }
670 }
671 if (tables.isFrenchSec())
672 {
673     if (preSecIgnore < secResult.length()) {
674         // If we've accumulated any secondary characters after the last base character,
675         // reverse them.
676         RBCollationTables.reverse(secResult, preSecIgnore, secResult.length());
677     }
678     // And now reverse the entire secResult to get French secondary ordering.
679     RBCollationTables.reverse(secResult, 0, secResult.length());
680 }
681 primResult.append((char)0);
682 secResult.append((char)0);
683 secResult.append(terResult.toString());
684 primResult.append(secResult.toString());
685
686 if (getStrength() == IDENTICAL) {
687     primResult.append((char)0);
688     int mode = getDecomposition();
689     if (mode == CANONICAL_DECOMPOSITION) {
690         primResult.append(Normalizer.normalize(source, Normalizer.Form.NFD));
691     } else if (mode == FULL_DECOMPOSITION) {
692         primResult.append(Normalizer.normalize(source, Normalizer.Form.NFKD));
693     } else {
694         primResult.append(source);
695     }
696 }
697 return new RuleBasedCollationKey(source, primResult.toString());
698 }
699
700 /**
701  * Standard override; no change in semantics.
702  */
703 public Object clone() {
704     // if we know we're not actually a subclass of RuleBasedCollator
705     // (this class really should have been made final), bypass
706     // Object.clone() and use our "copy constructor". This is faster.
707     if (getClass() == RuleBasedCollator.class) {
708         return new RuleBasedCollator(this);
709     }
710     else {

```

```

711         RuleBasedCollator result = (RuleBasedCollator) super.clone();
712         result.primResult = null;
713         result.secResult = null;
714         result.terResult = null;
715         result.sourceCursor = null;
716         result.targetCursor = null;
717         return result;
718     }
719 }
720
721 /**
722  * Compares the equality of two collation objects.
723  * @param obj the table-based collation object to be compared with this.
724  * @return true if the current table-based collation object is the same
725  * as the table-based collation object obj; false otherwise.
726  */
727 public boolean equals(Object obj) {
728     if (obj == null) return false;
729     if (!super.equals(obj)) return false; // super does class check
730     RuleBasedCollator other = (RuleBasedCollator) obj;
731     // all other non-transient information is also contained in rules.
732     return (getRules().equals(other.getRules()));
733 }
734
735 /**
736  * Generates the hash code for the table-based collation object
737  */
738 public int hashCode() {
739     return getRules().hashCode();
740 }
741
742 /**
743  * Allows CollationElementIterator access to the tables object
744  */
745 RBCollationTables getTables() {
746     return tables;
747 }
748
749 // =====
750 // private
751 // =====
752
753 final static int CHARINDEX = 0x70000000; // need look up in .commit()
754 final static int EXPANDCHARINDEX = 0x7E000000; // Expand index follows
755 final static int CONTRACTCHARINDEX = 0x7F000000; // contract indexes follow
756 final static int UNMAPPED = 0xFFFFFFFF;
757
758 private final static int COLLATIONKEYOFFSET = 1;
759
760 private RBCollationTables tables = null;
761
762 // Internal objects that are cached across calls so that they don't have to
763 // be created/destroyed on every call to compare() and getCollationKey()

```

```

764     private StringBuffer primResult = null;
765     private StringBuffer secResult = null;
766     private StringBuffer terResult = null;
767     private CollationElementIterator sourceCursor = null;
768     private CollationElementIterator targetCursor = null;
769 }

```

2.33 SimpleDateFormat.java

Secuencia 42: java/text/SimpleDateFormat.java

```

1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * (C) Copyright Taligent, Inc. 1996 - All Rights Reserved
28 * (C) Copyright IBM Corp. 1996-1998 - All Rights Reserved
29 *
30 * The original version of this source code and documentation is copyrighted
31 * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32 * materials are provided under terms of a License Agreement between Taligent
33 * and Sun. This technology is protected by multiple US and International
34 * patents. This notice and attribution to Taligent may not be removed.
35 * Taligent is a registered trademark of Taligent, Inc.
36 *
37 */
38
39 package java.text;
40
41 import java.io.IOException;
42 import java.io.InvalidObjectException;
43 import java.io.ObjectInputStream;

```

```

44 import static java.text.DateFormatSymbols.*;
45 import java.util.Calendar;
46 import java.util.Date;
47 import java.util.GregorianCalendar;
48 import java.util.Locale;
49 import java.util.Map;
50 import java.util.SimpleTimeZone;
51 import java.util.SortedMap;
52 import java.util.TimeZone;
53 import java.util.concurrent.ConcurrentHashMap;
54 import java.util.concurrent.ConcurrentMap;
55 import sun.util.calendar.CalendarUtils;
56 import sun.util.calendar.ZoneInfoFile;
57 import sun.util.locale.provider.LocaleProviderAdapter;
58
59 /**
60  * <code>SimpleDateFormat</code> is a concrete class for formatting and
61  * parsing dates in a locale-sensitive manner. It allows for formatting
62  * (date &rarr; text), parsing (text &rarr; date), and normalization.
63  *
64  * <p>
65  * <code>SimpleDateFormat</code> allows you to start by choosing
66  * any user-defined patterns for date-time formatting. However, you
67  * are encouraged to create a date-time formatter with either
68  * <code>getTimeInstance</code>, <code>getDateInstance</code>, or
69  * <code>getDateTimeInstance</code> in <code>DateFormat</code>. Each
70  * of these class methods can return a date/time formatter initialized
71  * with a default format pattern. You may modify the format pattern
72  * using the <code>applyPattern</code> methods as desired.
73  * For more information on using these methods, see
74  * {@link DateFormat}.
75  *
76  * <h3>Date and Time Patterns</h3>
77  * <p>
78  * Date and time formats are specified by <em>date and time pattern</em>
79  * strings.
80  * Within date and time pattern strings, unquoted letters from
81  * <code>'A'</code> to <code>'Z'</code> and from <code>'a'</code> to
82  * <code>'z'</code> are interpreted as pattern letters representing the
83  * components of a date or time string.
84  * Text can be quoted using single quotes (<code>'</code>) to avoid
85  * interpretation.
86  * <code>'"'"</code> represents a single quote.
87  * All other characters are not interpreted; they're simply copied into the
88  * output string during formatting or matched against the input string
89  * during parsing.
90  * <p>
91  * The following pattern letters are defined (all other characters from
92  * <code>'A'</code> to <code>'Z'</code> and from <code>'a'</code> to
93  * <code>'z'</code> are reserved):
94  * <blockquote>
95  * <table border=0 cellpadding=3 cellspacing=0 summary="Chart shows pattern letters, date/time
    component, presentation, and examples.">

```

```

96 *      <tr style="background-color: rgb(204, 204, 255);">
97 *          <th align=left>Letter
98 *          <th align=left>Date or Time Component
99 *          <th align=left>Presentation
100 *          <th align=left>Examples
101 *      <tr>
102 *          <td><code>G</code>
103 *          <td>Era designator
104 *          <td><a href="#text">Text</a>
105 *          <td><code>AD</code>
106 *      <tr style="background-color: rgb(238, 238, 255);">
107 *          <td><code>y</code>
108 *          <td>Year
109 *          <td><a href="#year">Year</a>
110 *          <td><code>1996</code>; <code>96</code>
111 *      <tr>
112 *          <td><code>Y</code>
113 *          <td>Week year
114 *          <td><a href="#year">Year</a>
115 *          <td><code>2009</code>; <code>09</code>
116 *      <tr style="background-color: rgb(238, 238, 255);">
117 *          <td><code>M</code>
118 *          <td>Month in year (context sensitive)
119 *          <td><a href="#month">Month</a>
120 *          <td><code>July</code>; <code>Jul</code>; <code>07</code>
121 *      <tr>
122 *          <td><code>L</code>
123 *          <td>Month in year (standalone form)
124 *          <td><a href="#month">Month</a>
125 *          <td><code>July</code>; <code>Jul</code>; <code>07</code>
126 *      <tr style="background-color: rgb(238, 238, 255);">
127 *          <td><code>w</code>
128 *          <td>Week in year
129 *          <td><a href="#number">Number</a>
130 *          <td><code>27</code>
131 *      <tr>
132 *          <td><code>W</code>
133 *          <td>Week in month
134 *          <td><a href="#number">Number</a>
135 *          <td><code>2</code>
136 *      <tr style="background-color: rgb(238, 238, 255);">
137 *          <td><code>D</code>
138 *          <td>Day in year
139 *          <td><a href="#number">Number</a>
140 *          <td><code>189</code>
141 *      <tr>
142 *          <td><code>d</code>
143 *          <td>Day in month
144 *          <td><a href="#number">Number</a>
145 *          <td><code>10</code>
146 *      <tr style="background-color: rgb(238, 238, 255);">
147 *          <td><code>F</code>
148 *          <td>Day of week in month

```



```

149 *      <td><a href="#number">Number</a>
150 *      <td><code>2</code>
151 *      <tr>
152 *      <td><code>E</code>
153 *      <td>Day name in week
154 *      <td><a href="#text">Text</a>
155 *      <td><code>Tuesday</code>; <code>Tue</code>
156 *      <tr style="background-color: rgb(238, 238, 255);">
157 *      <td><code>u</code>
158 *      <td>Day number of week (1 = Monday, ..., 7 = Sunday)
159 *      <td><a href="#number">Number</a>
160 *      <td><code>1</code>
161 *      <tr>
162 *      <td><code>a</code>
163 *      <td>Am/pm marker
164 *      <td><a href="#text">Text</a>
165 *      <td><code>PM</code>
166 *      <tr style="background-color: rgb(238, 238, 255);">
167 *      <td><code>H</code>
168 *      <td>Hour in day (0-23)
169 *      <td><a href="#number">Number</a>
170 *      <td><code>0</code>
171 *      <tr>
172 *      <td><code>k</code>
173 *      <td>Hour in day (1-24)
174 *      <td><a href="#number">Number</a>
175 *      <td><code>24</code>
176 *      <tr style="background-color: rgb(238, 238, 255);">
177 *      <td><code>K</code>
178 *      <td>Hour in am/pm (0-11)
179 *      <td><a href="#number">Number</a>
180 *      <td><code>0</code>
181 *      <tr>
182 *      <td><code>h</code>
183 *      <td>Hour in am/pm (1-12)
184 *      <td><a href="#number">Number</a>
185 *      <td><code>12</code>
186 *      <tr style="background-color: rgb(238, 238, 255);">
187 *      <td><code>m</code>
188 *      <td>Minute in hour
189 *      <td><a href="#number">Number</a>
190 *      <td><code>30</code>
191 *      <tr>
192 *      <td><code>s</code>
193 *      <td>Second in minute
194 *      <td><a href="#number">Number</a>
195 *      <td><code>55</code>
196 *      <tr style="background-color: rgb(238, 238, 255);">
197 *      <td><code>S</code>
198 *      <td>Millisecond
199 *      <td><a href="#number">Number</a>
200 *      <td><code>978</code>
201 *      <tr>

```

```

202 *      <td><code>z</code>
203 *      <td>Time zone
204 *      <td><a href="#timezone">General time zone</a>
205 *      <td><code>Pacific Standard Time</code>; <code>PST</code>; <code>GMT-08:00</code>
206 *      <tr style="background-color: rgb(238, 238, 255);">
207 *      <td><code>Z</code>
208 *      <td>Time zone
209 *      <td><a href="#rfc822timezone">RFC 822 time zone</a>
210 *      <td><code>-0800</code>
211 *      <tr>
212 *      <td><code>X</code>
213 *      <td>Time zone
214 *      <td><a href="#iso8601timezone">ISO 8601 time zone</a>
215 *      <td><code>-08</code>; <code>-0800</code>; <code>-08:00</code>
216 * </table>
217 * </blockquote>
218 * Pattern letters are usually repeated, as their number determines the
219 * exact presentation:
220 * <ul>
221 * <li><strong><a name="text">Text:</a></strong>
222 *     For formatting, if the number of pattern letters is 4 or more,
223 *     the full form is used; otherwise a short or abbreviated form
224 *     is used if available.
225 *     For parsing, both forms are accepted, independent of the number
226 *     of pattern letters.<br><br></li>
227 * <li><strong><a name="number">Number:</a></strong>
228 *     For formatting, the number of pattern letters is the minimum
229 *     number of digits, and shorter numbers are zero-padded to this amount.
230 *     For parsing, the number of pattern letters is ignored unless
231 *     it's needed to separate two adjacent fields.<br><br></li>
232 * <li><strong><a name="year">Year:</a></strong>
233 *     If the formatter's {@link #getCalendar() Calendar} is the Gregorian
234 *     calendar, the following rules are applied.<br>
235 *     <ul>
236 *     <li>For formatting, if the number of pattern letters is 2, the year
237 *         is truncated to 2 digits; otherwise it is interpreted as a
238 *         <a href="#number">number</a>.
239 *     <li>For parsing, if the number of pattern letters is more than 2,
240 *         the year is interpreted literally, regardless of the number of
241 *         digits. So using the pattern "MM/dd/yyyy", "01/11/12" parses to
242 *         Jan 11, 12 A.D.
243 *     <li>For parsing with the abbreviated year pattern ("y" or "yy"),
244 *         <code>SimpleDateFormat</code> must interpret the abbreviated year
245 *         relative to some century. It does this by adjusting dates to be
246 *         within 80 years before and 20 years after the time the <code>SimpleDateFormat</code>
247 *         instance is created. For example, using a pattern of "MM/dd/yy" and a
248 *         <code>SimpleDateFormat</code> instance created on Jan 1, 1997, the string
249 *         "01/11/12" would be interpreted as Jan 11, 2012 while the string "05/04/64"
250 *         would be interpreted as May 4, 1964.
251 *     During parsing, only strings consisting of exactly two digits, as defined by
252 *     {@link Character#isDigit(char)}, will be parsed into the default century.
253 *     Any other numeric string, such as a one digit string, a three or more digit
254 *     string, or a two digit string that isn't all digits (for example, "-1"), is

```

```

255 *      interpreted literally.  So "01/02/3" or "01/02/003" are parsed, using the
256 *      same pattern, as Jan 2, 3 AD.  Likewise, "01/02/-3" is parsed as Jan 2, 4 BC.
257 *
258 *      Otherwise, calendar system specific forms are applied.
259 *      For both formatting and parsing, if the number of pattern
260 *      letters is 4 or more, a calendar specific {@linkplain
261 *      Calendar#LONG long form} is used. Otherwise, a calendar
262 *      specific {@linkplain Calendar#SHORT short or abbreviated form}
263 *      is used.<br>
264 *      <br>
265 *      If week year {@code 'Y'} is specified and the {@linkplain
266 *      #getCalendar() calendar} doesn't support any <a
267 *      href="..util/GregorianCalendar.html#week_year"> week
268 *      years</a>, the calendar year ({@code 'y'}) is used instead. The
269 *      support of week years can be tested with a call to {@link
270 *      DateFormat#getCalendar() getCalendar()}.{@link
271 *      java.util.Calendar#isWeekDateSupported()
272 *      isWeekDateSupported()}.<br><br></li>
273 * <li><strong><a name="month">Month:</a></strong>
274 *      If the number of pattern letters is 3 or more, the month is
275 *      interpreted as <a href="#text">text</a>; otherwise,
276 *      it is interpreted as a <a href="#number">number</a>.<br>
277 *      <ul>
278 *      <li>Letter <em>M</em> produces context-sensitive month names, such as the
279 *      embedded form of names. If a {@code DateFormatSymbols} has been set
280 *      explicitly with constructor {@link #SimpleDateFormat(String,
281 *      DateFormatSymbols)} or method {@link
282 *      #setDateFormatSymbols(DateFormatSymbols)}, the month names given by
283 *      the {@code DateFormatSymbols} are used.</li>
284 *      <li>Letter <em>L</em> produces the standalone form of month names.</li>
285 *      </ul>
286 *      <br></li>
287 * <li><strong><a name="timezone">General time zone:</a></strong>
288 *      Time zones are interpreted as <a href="#text">text</a> if they have
289 *      names. For time zones representing a GMT offset value, the
290 *      following syntax is used:
291 *      <pre>
292 *      <a name="GMTOffsetTimeZone"><i>GMTOffsetTimeZone:</i></a>
293 *          <code>GMT</code> <i>Sign</i> <i>Hours</i> <code>:</code> <i>Minutes</i>
294 *      <i>Sign:</i> one of
295 *          <code>+ -</code>
296 *      <i>Hours:</i>
297 *          <i>Digit</i>
298 *          <i>Digit</i> <i>Digit</i>
299 *      <i>Minutes:</i>
300 *          <i>Digit</i> <i>Digit</i>
301 *      <i>Digit:</i> one of
302 *          <code>0 1 2 3 4 5 6 7 8 9</code></pre>
303 *      <i>Hours</i> must be between 0 and 23, and <i>Minutes</i> must be between
304 *      00 and 59. The format is locale independent and digits must be taken
305 *      from the Basic Latin block of the Unicode standard.
306 *      <p>For parsing, <a href="#rfc822timezone">RFC 822 time zones</a> are also
307 *      accepted.<br><br></li>

```

```

308 * <li><strong><a name="rfc822timezone">RFC 822 time zone:</a></strong>
309 *   For formatting, the RFC 822 4-digit time zone format is used:
310 *
311 *   <pre>
312 *   <i>RFC822TimeZone:</i>
313 *       <i>Sign</i> <i>TwoDigitHours</i> <i>Minutes</i>
314 *   <i>TwoDigitHours:</i>
315 *       <i>Digit Digit</i></pre>
316 *   <i>TwoDigitHours</i> must be between 00 and 23. Other definitions
317 *   are as for <a href="#timezone">general time zones</a>.
318 *
319 *   <p>For parsing, <a href="#timezone">general time zones</a> are also
320 *   accepted.
321 * </li><strong><a name="iso8601timezone">ISO 8601 Time zone:</a></strong>
322 *   The number of pattern letters designates the format for both formatting
323 *   and parsing as follows:
324 *   <pre>
325 *   <i>ISO8601TimeZone:</i>
326 *       <i>OneLetterISO8601TimeZone</i>
327 *       <i>TwoLetterISO8601TimeZone</i>
328 *       <i>ThreeLetterISO8601TimeZone</i>
329 *   <i>OneLetterISO8601TimeZone:</i>
330 *       <i>Sign</i> <i>TwoDigitHours</i>
331 *       {@code Z}
332 *   <i>TwoLetterISO8601TimeZone:</i>
333 *       <i>Sign</i> <i>TwoDigitHours</i> <i>Minutes</i>
334 *       {@code Z}
335 *   <i>ThreeLetterISO8601TimeZone:</i>
336 *       <i>Sign</i> <i>TwoDigitHours</i> {@code :} <i>Minutes</i>
337 *       {@code Z}</pre>
338 *   Other definitions are as for <a href="#timezone">general time zones</a> or
339 *   <a href="#rfc822timezone">RFC 822 time zones</a>.
340 *
341 *   <p>For formatting, if the offset value from GMT is 0, {@code "Z"} is
342 *   produced. If the number of pattern letters is 1, any fraction of an hour
343 *   is ignored. For example, if the pattern is {@code "X"} and the time zone is
344 *   {@code "GMT+05:30"}, {@code "+05"} is produced.
345 *
346 *   <p>For parsing, {@code "Z"} is parsed as the UTC time zone designator.
347 *   <a href="#timezone">General time zones</a> are <em>not</em> accepted.
348 *
349 *   <p>If the number of pattern letters is 4 or more, {@link
350 *   IllegalArgumentException} is thrown when constructing a {@code
351 *   SimpleDateFormat} or {@linkplain #applyPattern(String)} applying a
352 *   pattern}.
353 * </ul>
354 * <code>SimpleDateFormat</code> also supports <em>localized date and time
355 *   pattern</em> strings. In these strings, the pattern letters described above
356 *   may be replaced with other, locale dependent, pattern letters.
357 * <code>SimpleDateFormat</code> does not deal with the localization of text
358 *   other than the pattern letters; that's up to the client of the class.
359 *
360 * <h4>Examples</h4>

```

```

361 *
362 * The following examples show how date and time patterns are interpreted in
363 * the U.S. locale. The given date and time are 2001-07-04 12:08:56 local time
364 * in the U.S. Pacific Time time zone.
365 * <blockquote>
366 * <table border=0 cellspacing=3 cellpadding=0 summary="Examples of date and time patterns
    interpreted in the U.S. locale">
367 *     <tr style="background-color: rgb(204, 204, 255);">
368 *         <th align=left>Date and Time Pattern
369 *         <th align=left>Result
370 *     <tr>
371 *         <td><code>"yyyy.MM.dd G 'at' HH:mm:ss z"</code>
372 *         <td><code>2001.07.04 AD at 12:08:56 PDT</code>
373 *     <tr style="background-color: rgb(238, 238, 255);">
374 *         <td><code>"EEE, MMM d, 'yy"</code>
375 *         <td><code>Wed, Jul 4, '01</code>
376 *     <tr>
377 *         <td><code>"h:mm a"</code>
378 *         <td><code>12:08 PM</code>
379 *     <tr style="background-color: rgb(238, 238, 255);">
380 *         <td><code>"hh 'o''clock' a, zzzz"</code>
381 *         <td><code>12 o'clock PM, Pacific Daylight Time</code>
382 *     <tr>
383 *         <td><code>"K:mm a, z"</code>
384 *         <td><code>0:08 PM, PDT</code>
385 *     <tr style="background-color: rgb(238, 238, 255);">
386 *         <td><code>"yyyyy.MMMMM.dd GGG hh:mm aaa"</code>
387 *         <td><code>02001.July.04 AD 12:08 PM</code>
388 *     <tr>
389 *         <td><code>"EEE, d MMM yyyy HH:mm:ss Z"</code>
390 *         <td><code>Wed, 4 Jul 2001 12:08:56 -0700</code>
391 *     <tr style="background-color: rgb(238, 238, 255);">
392 *         <td><code>"yyMMddHHmmssZ"</code>
393 *         <td><code>010704120856-0700</code>
394 *     <tr>
395 *         <td><code>"yyyy-MM-dd'T'HH:mm:ss.SSSZ"</code>
396 *         <td><code>2001-07-04T12:08:56.235-0700</code>
397 *     <tr style="background-color: rgb(238, 238, 255);">
398 *         <td><code>"yyyy-MM-dd'T'HH:mm:ss.SSSXXX"</code>
399 *         <td><code>2001-07-04T12:08:56.235-07:00</code>
400 *     <tr>
401 *         <td><code>"YYYY-'W'ww-u"</code>
402 *         <td><code>2001-W27-3</code>
403 * </table>
404 * </blockquote>
405 *
406 * <h4><a name="synchronization">Synchronization</a></h4>
407 *
408 * <p>
409 * Date formats are not synchronized.
410 * It is recommended to create separate format instances for each thread.
411 * If multiple threads access a format concurrently, it must be synchronized
412 * externally.

```

```

413 *
414 * @see      <a href="https://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html
              ">Java Tutorial</a>
415 * @see      java.util.Calendar
416 * @see      java.util.TimeZone
417 * @see      DateFormat
418 * @see      DateFormatSymbols
419 * @author    Mark Davis, Chen-Lieh Huang, Alan Liu
420 */
421 public class SimpleDateFormat extends DateFormat {
422
423     // the official serial version ID which says cryptically
424     // which version we're compatible with
425     static final long serialVersionUID = 4774881970558875024L;
426
427     // the internal serial version which says which version was written
428     // - 0 (default) for version up to JDK 1.1.3
429     // - 1 for version from JDK 1.1.4, which includes a new field
430     static final int currentSerialVersion = 1;
431
432     /**
433      * The version of the serialized data on the stream. Possible values:
434      * <ul>
435      * <li><b>0</b> or not present on stream: JDK 1.1.3. This version
436      * has no <code>defaultCenturyStart</code> on stream.
437      * <li><b>1</b> JDK 1.1.4 or later. This version adds
438      * <code>defaultCenturyStart</code>.
439      * </ul>
440      * When streaming out this class, the most recent format
441      * and the highest allowable <code>serialVersionOnStream</code>
442      * is written.
443      * @serial
444      * @since JDK1.1.4
445      */
446     private int serialVersionOnStream = currentSerialVersion;
447
448     /**
449      * The pattern string of this formatter. This is always a non-localized
450      * pattern. May not be null. See class documentation for details.
451      * @serial
452      */
453     private String pattern;
454
455     /**
456      * Saved numberFormat and pattern.
457      * @see SimpleDateFormat#checkNegativeNumberExpression
458      */
459     transient private NumberFormat originalNumberFormat;
460     transient private String originalNumberPattern;
461
462     /**
463      * The minus sign to be used with format and parse.
464      */

```

```
465 transient private char minusSign = '-';
466
467 /**
468  * True when a negative sign follows a number.
469  * (True as default in Arabic.)
470  */
471 transient private boolean hasFollowingMinusSign = false;
472
473 /**
474  * True if standalone form needs to be used.
475  */
476 transient private boolean forceStandaloneForm = false;
477
478 /**
479  * The compiled pattern.
480  */
481 transient private char[] compiledPattern;
482
483 /**
484  * Tags for the compiled pattern.
485  */
486 private final static int TAG_QUOTE_ASCII_CHAR = 100;
487 private final static int TAG_QUOTE_CHARS = 101;
488
489 /**
490  * Locale dependent digit zero.
491  * @see #zeroPaddingNumber
492  * @see java.text.DecimalFormatSymbols#getZeroDigit
493  */
494 transient private char zeroDigit;
495
496 /**
497  * The symbols used by this formatter for week names, month names,
498  * etc. May not be null.
499  * @serial
500  * @see java.text.DateFormatSymbols
501  */
502 private DateFormatSymbols formatData;
503
504 /**
505  * We map dates with two-digit years into the century starting at
506  * <code>defaultCenturyStart</code>, which may be any date. May
507  * not be null.
508  * @serial
509  * @since JDK1.1.4
510  */
511 private Date defaultCenturyStart;
512
513 transient private int defaultCenturyStartYear;
514
515 private static final int MILLIS_PER_MINUTE = 60 * 1000;
516
517 // For time zones that have no names, use strings GMT+minutes and
```



```

518 // GMT-minutes. For instance, in France the time zone is GMT+60.
519 private static final String GMT = "GMT";
520
521 /**
522  * Cache NumberFormat instances with Locale key.
523  */
524 private static final ConcurrentMap<Locale, NumberFormat> cachedNumberFormatData
525     = new ConcurrentHashMap<>(3);
526
527 /**
528  * The Locale used to instantiate this
529  * <code>SimpleDateFormat</code>. The value may be null if this object
530  * has been created by an older <code>SimpleDateFormat</code> and
531  * deserialized.
532  *
533  * @serial
534  * @since 1.6
535  */
536 private Locale locale;
537
538 /**
539  * Indicates whether this <code>SimpleDateFormat</code> should use
540  * the DateFormatSymbols. If true, the format and parse methods
541  * use the DateFormatSymbols values. If false, the format and
542  * parse methods call Calendar.getDisplayName or
543  * Calendar.getDisplayNames.
544  */
545 transient boolean useDateFormatSymbols;
546
547 /**
548  * Constructs a <code>SimpleDateFormat</code> using the default pattern and
549  * date format symbols for the default
550  * {@link java.util.Locale.Category#FORMAT FORMAT} locale.
551  * <b>Note:</b> This constructor may not support all locales.
552  * For full coverage, use the factory methods in the {@link DateFormat}
553  * class.
554  */
555 public SimpleDateFormat() {
556     this("", Locale.getDefault(Locale.Category.FORMAT));
557     applyPatternImpl(LocaleProviderAdapter.getResourceBundleBased().getLocaleResources(locale)
558         .getDateTimePattern(SHORT, SHORT, calendar));
559 }
560
561 /**
562  * Constructs a <code>SimpleDateFormat</code> using the given pattern and
563  * the default date format symbols for the default
564  * {@link java.util.Locale.Category#FORMAT FORMAT} locale.
565  * <b>Note:</b> This constructor may not support all locales.
566  * For full coverage, use the factory methods in the {@link DateFormat}
567  * class.
568  * <p>This is equivalent to calling
569  * {@link #SimpleDateFormat(String, Locale)}
570  *     SimpleDateFormat(pattern, Locale.getDefault(Locale.Category.FORMAT))}.

```



```

571     *
572     * @see java.util.Locale#getDefault(java.util.Locale.Category)
573     * @see java.util.Locale.Category#FORMAT
574     * @param pattern the pattern describing the date and time format
575     * @exception NullPointerException if the given pattern is null
576     * @exception IllegalArgumentException if the given pattern is invalid
577     */
578     public SimpleDateFormat(String pattern)
579     {
580         this(pattern, Locale.getDefault(Locale.Category.FORMAT));
581     }
582
583     /**
584     * Constructs a SimpleDateFormat using the given pattern and
585     * the default date format symbols for the given locale.
586     * Note: This constructor may not support all locales.
587     * For full coverage, use the factory methods in the {@link DateFormat}
588     * class.
589     *
590     * @param pattern the pattern describing the date and time format
591     * @param locale the locale whose date format symbols should be used
592     * @exception NullPointerException if the given pattern or locale is null
593     * @exception IllegalArgumentException if the given pattern is invalid
594     */
595     public SimpleDateFormat(String pattern, Locale locale)
596     {
597         if (pattern == null || locale == null) {
598             throw new NullPointerException();
599         }
600
601         initializeCalendar(locale);
602         this.pattern = pattern;
603         this.formatData = DateFormatSymbols.getInstanceRef(locale);
604         this.locale = locale;
605         initialize(locale);
606     }
607
608     /**
609     * Constructs a SimpleDateFormat using the given pattern and
610     * date format symbols.
611     *
612     * @param pattern the pattern describing the date and time format
613     * @param formatSymbols the date format symbols to be used for formatting
614     * @exception NullPointerException if the given pattern or formatSymbols is null
615     * @exception IllegalArgumentException if the given pattern is invalid
616     */
617     public SimpleDateFormat(String pattern, DateFormatSymbols formatSymbols)
618     {
619         if (pattern == null || formatSymbols == null) {
620             throw new NullPointerException();
621         }
622
623         this.pattern = pattern;

```

```

624     this.formatData = (DateFormatSymbols) formatSymbols.clone();
625     this.locale = Locale.getDefault(Locale.Category.FORMAT);
626     initializeCalendar(this.locale);
627     initialize(this.locale);
628     useDateFormatSymbols = true;
629 }
630
631 /* Initialize compiledPattern and numberFormat fields */
632 private void initialize(Locale loc) {
633     // Verify and compile the given pattern.
634     compiledPattern = compile(pattern);
635
636     /* try the cache first */
637     numberFormat = cachedNumberFormatData.get(loc);
638     if (numberFormat == null) { /* cache miss */
639         numberFormat = NumberFormat.getIntegerInstance(loc);
640         numberFormat.setGroupingUsed(false);
641
642         /* update cache */
643         cachedNumberFormatData.putIfAbsent(loc, numberFormat);
644     }
645     numberFormat = (NumberFormat) numberFormat.clone();
646
647     initializeDefaultCentury();
648 }
649
650 private void initializeCalendar(Locale loc) {
651     if (calendar == null) {
652         assert loc != null;
653         // The format object must be constructed using the symbols for this zone.
654         // However, the calendar should use the current default TimeZone.
655         // If this is not contained in the locale zone strings, then the zone
656         // will be formatted using generic GMT+/-H:MM nomenclature.
657         calendar = Calendar.getInstance(TimeZone.getDefault(), loc);
658     }
659 }
660
661 /**
662  * Returns the compiled form of the given pattern. The syntax of
663  * the compiled pattern is:
664  * <blockquote>
665  * CompiledPattern:
666  *     EntryList
667  * EntryList:
668  *     Entry
669  *     EntryList Entry
670  * Entry:
671  *     TagField
672  *     TagField data
673  * TagField:
674  *     Tag Length
675  *     TaggedData
676  * Tag:

```

```

677 *     pattern_char_index
678 *     TAG_QUOTE_CHARS
679 * Length:
680 *     short_length
681 *     long_length
682 * TaggedData:
683 *     TAG_QUOTE_ASCII_CHAR ascii_char
684 *
685 * </blockquote>
686 *
687 * where `short_length' is an 8-bit unsigned integer between 0 and
688 * 254. `long_length' is a sequence of an 8-bit integer 255 and a
689 * 32-bit signed integer value which is split into upper and lower
690 * 16-bit fields in two char's. `pattern_char_index' is an 8-bit
691 * integer between 0 and 18. `ascii_char' is an 7-bit ASCII
692 * character value. `data' depends on its Tag value.
693 * <p>
694 * If Length is short_length, Tag and short_length are packed in a
695 * single char, as illustrated below.
696 * <blockquote>
697 *     char[0] = (Tag << 8) | short_length;
698 * </blockquote>
699 *
700 * If Length is long_length, Tag and 255 are packed in the first
701 * char and a 32-bit integer, as illustrated below.
702 * <blockquote>
703 *     char[0] = (Tag << 8) | 255;
704 *     char[1] = (char) (long_length >>> 16);
705 *     char[2] = (char) (long_length & 0xffff);
706 * </blockquote>
707 * <p>
708 * If Tag is a pattern_char_index, its Length is the number of
709 * pattern characters. For example, if the given pattern is
710 * "yyyy", Tag is 1 and Length is 4, followed by no data.
711 * <p>
712 * If Tag is TAG_QUOTE_CHARS, its Length is the number of char's
713 * following the TagField. For example, if the given pattern is
714 * "'o'clock'", Length is 7 followed by a char sequence of
715 * <code>o&nbsp;s; &nbsp;s;l&nbsp;s;o&nbsp;s;c&nbsp;s;k</code>.
716 * <p>
717 * TAG_QUOTE_ASCII_CHAR is a special tag and has an ASCII
718 * character in place of Length. For example, if the given pattern
719 * is "'o'", the TaggedData entry is
720 * <code>((TAG_QUOTE_ASCII_CHAR&nbsp;s;<&nbsp;s;8)&nbsp;s;|&nbsp;s;'o')</code>.
721 *
722 * @exception NullPointerException if the given pattern is null
723 * @exception IllegalArgumentException if the given pattern is invalid
724 */
725 private char[] compile(String pattern) {
726     int length = pattern.length();
727     boolean inQuote = false;
728     StringBuilder compiledCode = new StringBuilder(length * 2);
729     StringBuilder tmpBuffer = null;

```

```
730     int count = 0, tagcount = 0;
731     int lastTag = -1, prevTag = -1;
732
733     for (int i = 0; i < length; i++) {
734         char c = pattern.charAt(i);
735
736         if (c == '\\') {
737             // ' is treated as a single quote regardless of being
738             // in a quoted section.
739             if ((i + 1) < length) {
740                 c = pattern.charAt(i + 1);
741                 if (c == '\\') {
742                     i++;
743                     if (count != 0) {
744                         encode(lastTag, count, compiledCode);
745                         tagcount++;
746                         prevTag = lastTag;
747                         lastTag = -1;
748                         count = 0;
749                     }
750                     if (inQuote) {
751                         tmpBuffer.append(c);
752                     } else {
753                         compiledCode.append((char)(TAG_QUOTE_ASCII_CHAR << 8 | c));
754                     }
755                     continue;
756                 }
757             }
758             if (!inQuote) {
759                 if (count != 0) {
760                     encode(lastTag, count, compiledCode);
761                     tagcount++;
762                     prevTag = lastTag;
763                     lastTag = -1;
764                     count = 0;
765                 }
766                 if (tmpBuffer == null) {
767                     tmpBuffer = new StringBuilder(length);
768                 } else {
769                     tmpBuffer.setLength(0);
770                 }
771                 inQuote = true;
772             } else {
773                 int len = tmpBuffer.length();
774                 if (len == 1) {
775                     char ch = tmpBuffer.charAt(0);
776                     if (ch < 128) {
777                         compiledCode.append((char)(TAG_QUOTE_ASCII_CHAR << 8 | ch));
778                     } else {
779                         compiledCode.append((char)(TAG_QUOTE_CHARS << 8 | 1));
780                         compiledCode.append(ch);
781                     }
782                 } else {
```

```

783         encode(TAG_QUOTE_CHARS, len, compiledCode);
784         compiledCode.append(tmpBuffer);
785     }
786     inQuote = false;
787 }
788 continue;
789 }
790 if (inQuote) {
791     tmpBuffer.append(c);
792     continue;
793 }
794 if (!(c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')) {
795     if (count != 0) {
796         encode(lastTag, count, compiledCode);
797         tagcount++;
798         prevTag = lastTag;
799         lastTag = -1;
800         count = 0;
801     }
802     if (c < 128) {
803         // In most cases, c would be a delimiter, such as ':'.
804         compiledCode.append((char)(TAG_QUOTE_ASCII_CHAR << 8 | c));
805     } else {
806         // Take any contiguous non-ASCII alphabet characters and
807         // put them in a single TAG_QUOTE_CHARS.
808         int j;
809         for (j = i + 1; j < length; j++) {
810             char d = pattern.charAt(j);
811             if (d == '\\' || (d >= 'a' && d <= 'z' || d >= 'A' && d <= 'Z')) {
812                 break;
813             }
814         }
815         compiledCode.append((char)(TAG_QUOTE_CHARS << 8 | (j - i)));
816         for (; i < j; i++) {
817             compiledCode.append(pattern.charAt(i));
818         }
819         i--;
820     }
821     continue;
822 }
823
824 int tag;
825 if ((tag = DateFormatSymbols.patternChars.indexOf(c)) == -1) {
826     throw new IllegalArgumentException("Illegal pattern character " +
827                                     "'" + c + "'");
828 }
829 if (lastTag == -1 || lastTag == tag) {
830     lastTag = tag;
831     count++;
832     continue;
833 }
834 encode(lastTag, count, compiledCode);
835 tagcount++;

```

```

836         prevTag = lastTag;
837         lastTag = tag;
838         count = 1;
839     }
840
841     if (inQuote) {
842         throw new IllegalArgumentException("Unterminated quote");
843     }
844
845     if (count != 0) {
846         encode(lastTag, count, compiledCode);
847         tagcount++;
848         prevTag = lastTag;
849     }
850
851     forceStandaloneForm = (tagcount == 1 && prevTag == PATTERN_MONTH);
852
853     // Copy the compiled pattern to a char array
854     int len = compiledCode.length();
855     char[] r = new char[len];
856     compiledCode.getChars(0, len, r, 0);
857     return r;
858 }
859
860 /**
861  * Encodes the given tag and length and puts encoded char(s) into buffer.
862  */
863 private static void encode(int tag, int length, StringBuilder buffer) {
864     if (tag == PATTERN_ISO_ZONE && length >= 4) {
865         throw new IllegalArgumentException("invalid ISO 8601 format: length=" + length);
866     }
867     if (length < 255) {
868         buffer.append((char)(tag << 8 | length));
869     } else {
870         buffer.append((char)((tag << 8) | 0xff));
871         buffer.append((char)(length >>> 16));
872         buffer.append((char)(length & 0xffff));
873     }
874 }
875
876 /* Initialize the fields we use to disambiguate ambiguous years. Separate
877  * so we can call it from readObject().
878  */
879 private void initializeDefaultCentury() {
880     calendar.setTimeInMillis(System.currentTimeMillis());
881     calendar.add( Calendar.YEAR, -80 );
882     parseAmbiguousDatesAsAfter(calendar.getTime());
883 }
884
885 /* Define one-century window into which to disambiguate dates using
886  * two-digit years.
887  */
888 private void parseAmbiguousDatesAsAfter(Date startDate) {

```

```

889     defaultCenturyStart = startDate;
890     calendar.setTime(startDate);
891     defaultCenturyStartYear = calendar.get(Calendar.YEAR);
892 }
893
894 /**
895  * Sets the 100-year period 2-digit years will be interpreted as being in
896  * to begin on the date the user specifies.
897  *
898  * @param startDate During parsing, two digit years will be placed in the range
899  * <code>startDate</code> to <code>startDate + 100 years</code>.
900  * @see #get2DigitYearStart
901  * @since 1.2
902  */
903 public void set2DigitYearStart(Date startDate) {
904     parseAmbiguousDatesAsAfter(new Date(startDate.getTime()));
905 }
906
907 /**
908  * Returns the beginning date of the 100-year period 2-digit years are interpreted
909  * as being within.
910  *
911  * @return the start of the 100-year period into which two digit years are
912  * parsed
913  * @see #set2DigitYearStart
914  * @since 1.2
915  */
916 public Date get2DigitYearStart() {
917     return (Date) defaultCenturyStart.clone();
918 }
919
920 /**
921  * Formats the given <code>Date</code> into a date/time string and appends
922  * the result to the given <code>StringBuffer</code>.
923  *
924  * @param date the date-time value to be formatted into a date-time string.
925  * @param toAppendTo where the new date-time text is to be appended.
926  * @param pos the formatting position. On input: an alignment field,
927  * if desired. On output: the offsets of the alignment field.
928  * @return the formatted date-time string.
929  * @exception NullPointerException if the given {@code date} is {@code null}.
930  */
931 @Override
932 public StringBuffer format(Date date, StringBuffer toAppendTo,
933     FieldPosition pos)
934 {
935     pos.beginIndex = pos.endIndex = 0;
936     return format(date, toAppendTo, pos.getFieldDelegate());
937 }
938
939 // Called from Format after creating a FieldDelegate
940 private StringBuffer format(Date date, StringBuffer toAppendTo,
941     FieldDelegate delegate) {

```

```

942     // Convert input date to time field list
943     calendar.setTime(date);
944
945     boolean useDateFormatSymbols = useDateFormatSymbols();
946
947     for (int i = 0; i < compiledPattern.length; ) {
948         int tag = compiledPattern[i] >>> 8;
949         int count = compiledPattern[i++] & 0xff;
950         if (count == 255) {
951             count = compiledPattern[i++] << 16;
952             count |= compiledPattern[i++];
953         }
954
955         switch (tag) {
956             case TAG_QUOTE_ASCII_CHAR:
957                 toAppendTo.append((char)count);
958                 break;
959
960             case TAG_QUOTE_CHARS:
961                 toAppendTo.append(compiledPattern, i, count);
962                 i += count;
963                 break;
964
965             default:
966                 subFormat(tag, count, delegate, toAppendTo, useDateFormatSymbols);
967                 break;
968         }
969     }
970     return toAppendTo;
971 }
972
973 /**
974  * Formats an Object producing an AttributedCharacterIterator.
975  * You can use the returned AttributedCharacterIterator
976  * to build the resulting String, as well as to determine information
977  * about the resulting String.
978  * <p>
979  * Each attribute key of the AttributedCharacterIterator will be of type
980  * DateFormat.Field, with the corresponding attribute value
981  * being the same as the attribute key.
982  *
983  * @exception NullPointerException if obj is null.
984  * @exception IllegalArgumentException if the Format cannot format the
985  *         given object, or if the Format's pattern string is invalid.
986  * @param obj The object to format
987  * @return AttributedCharacterIterator describing the formatted value.
988  * @since 1.4
989  */
990 @Override
991 public AttributedCharacterIterator formatToCharacterIterator(Object obj) {
992     StringBuffer sb = new StringBuffer();
993     CharacterIteratorFieldDelegate delegate = new
994         CharacterIteratorFieldDelegate();

```



```
995
996     if (obj instanceof Date) {
997         format((Date)obj, sb, delegate);
998     }
999     else if (obj instanceof Number) {
1000         format(new Date(((Number)obj).longValue()), sb, delegate);
1001     }
1002     else if (obj == null) {
1003         throw new NullPointerException(
1004             "formatToCharacterIterator must be passed non-null object");
1005     }
1006     else {
1007         throw new IllegalArgumentException(
1008             "Cannot format given Object as a Date");
1009     }
1010     return delegate.getIterator(sb.toString());
1011 }
1012
1013 // Map index into pattern character string to Calendar field number
1014 private static final int[] PATTERN_INDEX_TO_CALENDAR_FIELD = {
1015     Calendar.ERA,
1016     Calendar.YEAR,
1017     Calendar.MONTH,
1018     Calendar.DATE,
1019     Calendar.HOUR_OF_DAY,
1020     Calendar.HOUR_OF_DAY,
1021     Calendar.MINUTE,
1022     Calendar.SECOND,
1023     Calendar.MILLISECOND,
1024     Calendar.DAY_OF_WEEK,
1025     Calendar.DAY_OF_YEAR,
1026     Calendar.DAY_OF_WEEK_IN_MONTH,
1027     Calendar.WEEK_OF_YEAR,
1028     Calendar.WEEK_OF_MONTH,
1029     Calendar.AM_PM,
1030     Calendar.HOUR,
1031     Calendar.HOUR,
1032     Calendar.ZONE_OFFSET,
1033     Calendar.ZONE_OFFSET,
1034     CalendarBuilder.WEEK_YEAR, // Pseudo Calendar field
1035     CalendarBuilder.ISO_DAY_OF_WEEK, // Pseudo Calendar field
1036     Calendar.ZONE_OFFSET,
1037     Calendar.MONTH
1038 };
1039
1040 // Map index into pattern character string to DateFormat field number
1041 private static final int[] PATTERN_INDEX_TO_DATE_FORMAT_FIELD = {
1042     DateFormat.ERA_FIELD,
1043     DateFormat.YEAR_FIELD,
1044     DateFormat.MONTH_FIELD,
1045     DateFormat.DATE_FIELD,
1046     DateFormat.HOUR_OF_DAY1_FIELD,
1047     DateFormat.HOUR_OF_DAY0_FIELD,
```

```
1048     DateFormat.MINUTE_FIELD,
1049     DateFormat.SECOND_FIELD,
1050     DateFormat.MILLISECOND_FIELD,
1051     DateFormat.DAY_OF_WEEK_FIELD,
1052     DateFormat.DAY_OF_YEAR_FIELD,
1053     DateFormat.DAY_OF_WEEK_IN_MONTH_FIELD,
1054     DateFormat.WEEK_OF_YEAR_FIELD,
1055     DateFormat.WEEK_OF_MONTH_FIELD,
1056     DateFormat.AM_PM_FIELD,
1057     DateFormat.HOUR1_FIELD,
1058     DateFormat.HOURO_FIELD,
1059     DateFormat.TIMEZONE_FIELD,
1060     DateFormat.TIMEZONE_FIELD,
1061     DateFormat.YEAR_FIELD,
1062     DateFormat.DAY_OF_WEEK_FIELD,
1063     DateFormat.TIMEZONE_FIELD,
1064     DateFormat.MONTH_FIELD
1065 };
1066
1067 // Maps from DecimalFormatSymbols index to Field constant
1068 private static final Field[] PATTERN_INDEX_TO_DATE_FORMAT_FIELD_ID = {
1069     Field.ERA,
1070     Field.YEAR,
1071     Field.MONTH,
1072     Field.DAY_OF_MONTH,
1073     Field.HOUR_OF_DAY1,
1074     Field.HOUR_OF_DAY0,
1075     Field.MINUTE,
1076     Field.SECOND,
1077     Field.MILLISECOND,
1078     Field.DAY_OF_WEEK,
1079     Field.DAY_OF_YEAR,
1080     Field.DAY_OF_WEEK_IN_MONTH,
1081     Field.WEEK_OF_YEAR,
1082     Field.WEEK_OF_MONTH,
1083     Field.AM_PM,
1084     Field.HOUR1,
1085     Field.HOURO,
1086     Field.TIME_ZONE,
1087     Field.TIME_ZONE,
1088     Field.YEAR,
1089     Field.DAY_OF_WEEK,
1090     Field.TIME_ZONE,
1091     Field.MONTH
1092 };
1093
1094 /**
1095  * Private member function that does the real date/time formatting.
1096  */
1097 private void subFormat(int patternCharIndex, int count,
1098     FieldDelegate delegate, StringBuffer buffer,
1099     boolean useDateFormatSymbols)
1100 {
```

```

1101     int    maxIntCount = Integer.MAX_VALUE;
1102     String current = null;
1103     int    beginOffset = buffer.length();
1104
1105     int field = PATTERN_INDEX_TO_CALENDAR_FIELD[patternCharIndex];
1106     int value;
1107     if (field == CalendarBuilder.WEEK_YEAR) {
1108         if (calendar.isWeekDateSupported()) {
1109             value = calendar.getWeekYear();
1110         } else {
1111             // use calendar year 'y' instead
1112             patternCharIndex = PATTERN_YEAR;
1113             field = PATTERN_INDEX_TO_CALENDAR_FIELD[patternCharIndex];
1114             value = calendar.get(field);
1115         }
1116     } else if (field == CalendarBuilder.ISO_DAY_OF_WEEK) {
1117         value = CalendarBuilder.toISODayOfWeek(calendar.get(Calendar.DAY_OF_WEEK));
1118     } else {
1119         value = calendar.get(field);
1120     }
1121
1122     int style = (count >= 4) ? Calendar.LONG : Calendar.SHORT;
1123     if (!useDateFormatSymbols && field < Calendar.ZONE_OFFSET
1124         && patternCharIndex != PATTERN_MONTH_STANDALONE) {
1125         current = calendar.getDisplayName(field, style, locale);
1126     }
1127
1128     // Note: zeroPaddingNumber() assumes that maxDigits is either
1129     // 2 or maxIntCount. If we make any changes to this,
1130     // zeroPaddingNumber() must be fixed.
1131
1132     switch (patternCharIndex) {
1133     case PATTERN_ERA: // 'G'
1134         if (useDateFormatSymbols) {
1135             String[] eras = formatData.getEras();
1136             if (value < eras.length) {
1137                 current = eras[value];
1138             }
1139         }
1140         if (current == null) {
1141             current = "";
1142         }
1143         break;
1144
1145     case PATTERN_WEEK_YEAR: // 'Y'
1146     case PATTERN_YEAR:      // 'y'
1147         if (calendar instanceof GregorianCalendar) {
1148             if (count != 2) {
1149                 zeroPaddingNumber(value, count, maxIntCount, buffer);
1150             } else {
1151                 zeroPaddingNumber(value, 2, 2, buffer);
1152             } // clip 1996 to 96
1153         } else {

```

```
1154         if (current == null) {
1155             zeroPaddingNumber(value, style == Calendar.LONG ? 1 : count,
1156                             maxIntCount, buffer);
1157         }
1158     }
1159     break;
1160
1161     case PATTERN_MONTH: // 'M' (context seinsive)
1162         if (useDateFormatSymbols) {
1163             String[] months;
1164             if (count >= 4) {
1165                 months = formatData.getMonths();
1166                 current = months[value];
1167             } else if (count == 3) {
1168                 months = formatData.getShortMonths();
1169                 current = months[value];
1170             }
1171         } else {
1172             if (count < 3) {
1173                 current = null;
1174             } else if (forceStandaloneForm) {
1175                 current = calendar.getDisplayName(field, style | 0x8000, locale);
1176                 if (current == null) {
1177                     current = calendar.getDisplayName(field, style, locale);
1178                 }
1179             }
1180         }
1181         if (current == null) {
1182             zeroPaddingNumber(value+1, count, maxIntCount, buffer);
1183         }
1184         break;
1185
1186     case PATTERN_MONTH_STANDALONE: // 'L'
1187         assert current == null;
1188         if (locale == null) {
1189             String[] months;
1190             if (count >= 4) {
1191                 months = formatData.getMonths();
1192                 current = months[value];
1193             } else if (count == 3) {
1194                 months = formatData.getShortMonths();
1195                 current = months[value];
1196             }
1197         } else {
1198             if (count >= 3) {
1199                 current = calendar.getDisplayName(field, style | 0x8000, locale);
1200             }
1201         }
1202         if (current == null) {
1203             zeroPaddingNumber(value+1, count, maxIntCount, buffer);
1204         }
1205         break;
1206
```

```
1207     case PATTERN_HOUR_OF_DAY1: // 'k' 1-based. eg, 23:59 + 1 hour ==> 24:59
1208         if (current == null) {
1209             if (value == 0) {
1210                 zeroPaddingNumber(calendar.getMaximum(Calendar.HOUR_OF_DAY) + 1,
1211                                     count, maxIntCount, buffer);
1212             } else {
1213                 zeroPaddingNumber(value, count, maxIntCount, buffer);
1214             }
1215         }
1216         break;
1217
1218     case PATTERN_DAY_OF_WEEK: // 'E'
1219         if (useDateFormatSymbols) {
1220             String[] weekdays;
1221             if (count >= 4) {
1222                 weekdays = formatData.getWeekdays();
1223                 current = weekdays[value];
1224             } else { // count < 4, use abbreviated form if exists
1225                 weekdays = formatData.getShortWeekdays();
1226                 current = weekdays[value];
1227             }
1228         }
1229         break;
1230
1231     case PATTERN_AM_PM: // 'a'
1232         if (useDateFormatSymbols) {
1233             String[] ampm = formatData.getAmPmStrings();
1234             current = ampm[value];
1235         }
1236         break;
1237
1238     case PATTERN_HOUR1: // 'h' 1-based. eg, 11PM + 1 hour ==> 12 AM
1239         if (current == null) {
1240             if (value == 0) {
1241                 zeroPaddingNumber(calendar.getLeastMaximum(Calendar.HOUR) + 1,
1242                                     count, maxIntCount, buffer);
1243             } else {
1244                 zeroPaddingNumber(value, count, maxIntCount, buffer);
1245             }
1246         }
1247         break;
1248
1249     case PATTERN_ZONE_NAME: // 'z'
1250         if (current == null) {
1251             if (formatData.locale == null || formatData.isZoneStringsSet) {
1252                 int zoneIndex =
1253                     formatData.getZoneIndex(calendar.getTimeZone().getID());
1254                 if (zoneIndex == -1) {
1255                     value = calendar.get(Calendar.ZONE_OFFSET) +
1256                             calendar.get(Calendar.DST_OFFSET);
1257                     buffer.append(ZoneInfoFile.toCustomID(value));
1258                 } else {
1259                     int index = (calendar.get(Calendar.DST_OFFSET) == 0) ? 1: 3;
```

```

1260         if (count < 4) {
1261             // Use the short name
1262             index++;
1263         }
1264         String[] [] zoneStrings = formatData.getZoneStringsWrapper();
1265         buffer.append(zoneStrings[zoneIndex][index]);
1266     }
1267 } else {
1268     TimeZone tz = calendar.getTimeZone();
1269     boolean daylight = (calendar.get(Calendar.DST_OFFSET) != 0);
1270     int tzstyle = (count < 4 ? TimeZone.SHORT : TimeZone.LONG);
1271     buffer.append(tz.getDisplayName(daylight, tzstyle, formatData.locale));
1272 }
1273 }
1274 break;
1275
1276 case PATTERN_ZONE_VALUE: // 'Z' ("-/+hhmm" form)
1277     value = (calendar.get(Calendar.ZONE_OFFSET) +
1278             calendar.get(Calendar.DST_OFFSET)) / 60000;
1279
1280     int width = 4;
1281     if (value >= 0) {
1282         buffer.append('+');
1283     } else {
1284         width++;
1285     }
1286
1287     int num = (value / 60) * 100 + (value % 60);
1288     CalendarUtils.printf0d(buffer, num, width);
1289     break;
1290
1291 case PATTERN_ISO_ZONE: // 'X'
1292     value = calendar.get(Calendar.ZONE_OFFSET)
1293             + calendar.get(Calendar.DST_OFFSET);
1294
1295     if (value == 0) {
1296         buffer.append('Z');
1297         break;
1298     }
1299
1300     value /= 60000;
1301     if (value >= 0) {
1302         buffer.append('+');
1303     } else {
1304         buffer.append('-');
1305         value = -value;
1306     }
1307
1308     CalendarUtils.printf0d(buffer, value / 60, 2);
1309     if (count == 1) {
1310         break;
1311     }
1312

```

```

1313         if (count == 3) {
1314             buffer.append(':');
1315         }
1316         CalendarUtils.sprintf0d(buffer, value % 60, 2);
1317         break;
1318
1319         default:
1320             // case PATTERN_DAY_OF_MONTH:           // 'd'
1321             // case PATTERN_HOUR_OF_DAY0:           // 'H' 0-based. eg, 23:59 + 1 hour ==> 00:59
1322             // case PATTERN_MINUTE:                 // 'm'
1323             // case PATTERN_SECOND:                 // 's'
1324             // case PATTERN_MILLISECOND:            // 'S'
1325             // case PATTERN_DAY_OF_YEAR:            // 'D'
1326             // case PATTERN_DAY_OF_WEEK_IN_MONTH:   // 'F'
1327             // case PATTERN_WEEK_OF_YEAR:           // 'w'
1328             // case PATTERN_WEEK_OF_MONTH:          // 'W'
1329             // case PATTERN_HOURO:                  // 'K' eg, 11PM + 1 hour ==> 0 AM
1330             // case PATTERN_ISO_DAY_OF_WEEK:        // 'u' pseudo field, Monday = 1, ..., Sunday = 7
1331             if (current == null) {
1332                 zeroPaddingNumber(value, count, maxIntCount, buffer);
1333             }
1334             break;
1335         } // switch (patternCharIndex)
1336
1337         if (current != null) {
1338             buffer.append(current);
1339         }
1340
1341         int fieldID = PATTERN_INDEX_TO_DATE_FORMAT_FIELD[patternCharIndex];
1342         Field f = PATTERN_INDEX_TO_DATE_FORMAT_FIELD_ID[patternCharIndex];
1343
1344         delegate.formatted(fieldID, f, f, beginOffset, buffer.length(), buffer);
1345     }
1346
1347     /**
1348      * Formats a number with the specified minimum and maximum number of digits.
1349      */
1350     private void zeroPaddingNumber(int value, int minDigits, int maxDigits, StringBuffer buffer)
1351     {
1352         // Optimization for 1, 2 and 4 digit numbers. This should
1353         // cover most cases of formatting date/time related items.
1354         // Note: This optimization code assumes that maxDigits is
1355         // either 2 or Integer.MAX_VALUE (maxIntCount in format()).
1356         try {
1357             if (zeroDigit == 0) {
1358                 zeroDigit = ((DecimalFormat)numberFormat).getDecimalFormatSymbols().getZeroDigit();
1359             }
1360             if (value >= 0) {
1361                 if (value < 100 && minDigits >= 1 && minDigits <= 2) {
1362                     if (value < 10) {
1363                         if (minDigits == 2) {
1364                             buffer.append(zeroDigit);
1365                         }

```

```

1366         buffer.append((char)(zeroDigit + value));
1367     } else {
1368         buffer.append((char)(zeroDigit + value / 10));
1369         buffer.append((char)(zeroDigit + value % 10));
1370     }
1371     return;
1372 } else if (value >= 1000 && value < 10000) {
1373     if (minDigits == 4) {
1374         buffer.append((char)(zeroDigit + value / 1000));
1375         value %= 1000;
1376         buffer.append((char)(zeroDigit + value / 100));
1377         value %= 100;
1378         buffer.append((char)(zeroDigit + value / 10));
1379         buffer.append((char)(zeroDigit + value % 10));
1380         return;
1381     }
1382     if (minDigits == 2 && maxDigits == 2) {
1383         zeroPaddingNumber(value % 100, 2, 2, buffer);
1384         return;
1385     }
1386 }
1387 }
1388 } catch (Exception e) {
1389 }
1390
1391 numberFormat.setMinimumIntegerDigits(minDigits);
1392 numberFormat.setMaximumIntegerDigits(maxDigits);
1393 numberFormat.format((long)value, buffer, DontCareFieldPosition.INSTANCE);
1394 }
1395
1396
1397 /**
1398  * Parses text from a string to produce a <code>Date</code>.
1399  * <p>
1400  * The method attempts to parse text starting at the index given by
1401  * <code>pos</code>.
1402  * If parsing succeeds, then the index of <code>pos</code> is updated
1403  * to the index after the last character used (parsing does not necessarily
1404  * use all characters up to the end of the string), and the parsed
1405  * date is returned. The updated <code>pos</code> can be used to
1406  * indicate the starting point for the next call to this method.
1407  * If an error occurs, then the index of <code>pos</code> is not
1408  * changed, the error index of <code>pos</code> is set to the index of
1409  * the character where the error occurred, and null is returned.
1410  *
1411  * <p>This parsing operation uses the {@link DateFormat#calendar
1412  * calendar} to produce a {@code Date}. All of the {@code
1413  * calendar}'s date-time fields are {@link plain Calendar#clear()}
1414  * cleared} before parsing, and the {@code calendar}'s default
1415  * values of the date-time fields are used for any missing
1416  * date-time information. For example, the year value of the
1417  * parsed {@code Date} is 1970 with {@link GregorianCalendar} if
1418  * no year value is given from the parsing operation. The {@code

```



```

1419 * TimeZone} value may be overwritten, depending on the given
1420 * pattern and the time zone value in {@code text}. Any {@code
1421 * TimeZone} value that has previously been set by a call to
1422 * {@link #setTimeZone(java.util.TimeZone) setTimeZone} may need
1423 * to be restored for further operations.
1424 *
1425 * @param text A String, part of which should be parsed.
1426 * @param pos A ParsePosition object with index and error
1427 *            index information as described above.
1428 * @return A Date parsed from the string. In case of
1429 *         error, returns null.
1430 * @exception NullPointerException if text or pos is null.
1431 */
1432 @Override
1433 public Date parse(String text, ParsePosition pos)
1434 {
1435     checkNegativeNumberExpression();
1436
1437     int start = pos.index;
1438     int oldStart = start;
1439     int textLength = text.length();
1440
1441     boolean[] ambiguousYear = {false};
1442
1443     CalendarBuilder calb = new CalendarBuilder();
1444
1445     for (int i = 0; i < compiledPattern.length; ) {
1446         int tag = compiledPattern[i] >>> 8;
1447         int count = compiledPattern[i++] & 0xff;
1448         if (count == 255) {
1449             count = compiledPattern[i++] << 16;
1450             count |= compiledPattern[i++];
1451         }
1452
1453         switch (tag) {
1454             case TAG_QUOTE_ASCII_CHAR:
1455                 if (start >= textLength || text.charAt(start) != (char)count) {
1456                     pos.index = oldStart;
1457                     pos.errorIndex = start;
1458                     return null;
1459                 }
1460                 start++;
1461                 break;
1462
1463             case TAG_QUOTE_CHARS:
1464                 while (count-- > 0) {
1465                     if (start >= textLength || text.charAt(start) != compiledPattern[i++]) {
1466                         pos.index = oldStart;
1467                         pos.errorIndex = start;
1468                         return null;
1469                     }
1470                     start++;
1471                 }

```

```

1472         break;
1473
1474     default:
1475         // Peek the next pattern to determine if we need to
1476         // obey the number of pattern letters for
1477         // parsing. It's required when parsing contiguous
1478         // digit text (e.g., "20010704") with a pattern which
1479         // has no delimiters between fields, like "yyyyMMdd".
1480         boolean obeyCount = false;
1481
1482         // In Arabic, a minus sign for a negative number is put after
1483         // the number. Even in another locale, a minus sign can be
1484         // put after a number using DateFormat.setNumberFormat().
1485         // If both the minus sign and the field-delimiter are '-',
1486         // subParse() needs to determine whether a '-' after a number
1487         // in the given text is a delimiter or is a minus sign for the
1488         // preceding number. We give subParse() a clue based on the
1489         // information in compiledPattern.
1490         boolean useFollowingMinusSignAsDelimiter = false;
1491
1492         if (i < compiledPattern.length) {
1493             int nextTag = compiledPattern[i] >>> 8;
1494             if (!(nextTag == TAG_QUOTE_ASCII_CHAR ||
1495                 nextTag == TAG_QUOTE_CHARS)) {
1496                 obeyCount = true;
1497             }
1498
1499             if (hasFollowingMinusSign &&
1500                 (nextTag == TAG_QUOTE_ASCII_CHAR ||
1501                  nextTag == TAG_QUOTE_CHARS)) {
1502                 int c;
1503                 if (nextTag == TAG_QUOTE_ASCII_CHAR) {
1504                     c = compiledPattern[i] & 0xff;
1505                 } else {
1506                     c = compiledPattern[i+1];
1507                 }
1508
1509                 if (c == minusSign) {
1510                     useFollowingMinusSignAsDelimiter = true;
1511                 }
1512             }
1513         }
1514         start = subParse(text, start, tag, count, obeyCount,
1515                         ambiguousYear, pos,
1516                         useFollowingMinusSignAsDelimiter, calb);
1517         if (start < 0) {
1518             pos.index = oldStart;
1519             return null;
1520         }
1521     }
1522 }
1523
1524 // At this point the fields of Calendar have been set.  Calendar

```

```

1525     // will fill in default values for missing fields when the time
1526     // is computed.
1527
1528     pos.index = start;
1529
1530     Date parsedDate;
1531     try {
1532         parsedDate = calb.establish(calendar).getTime();
1533         // If the year value is ambiguous,
1534         // then the two-digit year == the default start year
1535         if (ambiguousYear[0]) {
1536             if (parsedDate.before(defaultCenturyStart)) {
1537                 parsedDate = calb.addYear(100).establish(calendar).getTime();
1538             }
1539         }
1540     }
1541     // An IllegalArgumentException will be thrown by Calendar.getTime()
1542     // if any fields are out of range, e.g., MONTH == 17.
1543     catch (IllegalArgumentException e) {
1544         pos.errorIndex = start;
1545         pos.index = oldStart;
1546         return null;
1547     }
1548
1549     return parsedDate;
1550 }
1551
1552 /**
1553  * Private code-size reduction function used by subParse.
1554  * @param text the time text being parsed.
1555  * @param start where to start parsing.
1556  * @param field the date field being parsed.
1557  * @param data the string array to parsed.
1558  * @return the new start position if matching succeeded; a negative number
1559  *         indicating matching failure, otherwise.
1560  */
1561 private int matchString(String text, int start, int field, String[] data, CalendarBuilder calb)
1562 {
1563     int i = 0;
1564     int count = data.length;
1565
1566     if (field == Calendar.DAY_OF_WEEK) {
1567         i = 1;
1568     }
1569
1570     // There may be multiple strings in the data[] array which begin with
1571     // the same prefix (e.g., Cerven and Cervenec (June and July) in Czech).
1572     // We keep track of the longest match, and return that. Note that this
1573     // unfortunately requires us to test all array elements.
1574     int bestMatchLength = 0, bestMatch = -1;
1575     for (; i < count; ++i)
1576     {
1577         int length = data[i].length();

```

```
1578         // Always compare if we have no match yet; otherwise only compare
1579         // against potentially better matches (longer strings).
1580         if (length > bestMatchLength &&
1581             text.regionMatches(true, start, data[i], 0, length))
1582         {
1583             bestMatch = i;
1584             bestMatchLength = length;
1585         }
1586     }
1587     if (bestMatch >= 0)
1588     {
1589         calb.set(field, bestMatch);
1590         return start + bestMatchLength;
1591     }
1592     return -start;
1593 }
1594
1595 /**
1596  * Performs the same thing as matchString(String, int, int,
1597  * String[]). This method takes a Map<String, Integer> instead of
1598  * String[].
1599  */
1600 private int matchString(String text, int start, int field,
1601                         Map<String,Integer> data, CalendarBuilder calb) {
1602     if (data != null) {
1603         // TODO: make this default when it's in the spec.
1604         if (data instanceof SortedMap) {
1605             for (String name : data.keySet()) {
1606                 if (text.regionMatches(true, start, name, 0, name.length())) {
1607                     calb.set(field, data.get(name));
1608                     return start + name.length();
1609                 }
1610             }
1611             return -start;
1612         }
1613
1614         String bestMatch = null;
1615
1616         for (String name : data.keySet()) {
1617             int length = name.length();
1618             if (bestMatch == null || length > bestMatch.length()) {
1619                 if (text.regionMatches(true, start, name, 0, length)) {
1620                     bestMatch = name;
1621                 }
1622             }
1623         }
1624
1625         if (bestMatch != null) {
1626             calb.set(field, data.get(bestMatch));
1627             return start + bestMatch.length();
1628         }
1629     }
1630     return -start;
}
```

```

1631     }
1632
1633     private int matchZoneString(String text, int start, String[] zoneNames) {
1634         for (int i = 1; i <= 4; ++i) {
1635             // Checking long and short zones [1 & 2],
1636             // and long and short daylight [3 & 4].
1637             String zoneName = zoneNames[i];
1638             if (text.regionMatches(true, start,
1639                                     zoneName, 0, zoneName.length())) {
1640                 return i;
1641             }
1642         }
1643         return -1;
1644     }
1645
1646     private boolean matchDSTString(String text, int start, int zoneIndex, int standardIndex,
1647                                     String[] [] zoneStrings) {
1648         int index = standardIndex + 2;
1649         String zoneName = zoneStrings[zoneIndex][index];
1650         if (text.regionMatches(true, start,
1651                                 zoneName, 0, zoneName.length())) {
1652             return true;
1653         }
1654         return false;
1655     }
1656
1657     /**
1658     * find time zone 'text' matched zoneStrings and set to internal
1659     * calendar.
1660     */
1661     private int subParseZoneString(String text, int start, CalendarBuilder calb) {
1662         boolean useSameName = false; // true if standard and daylight time use the same
            abbreviation.
1663         TimeZone currentTimeZone = getTimeZone();
1664
1665         // At this point, check for named time zones by looking through
1666         // the locale data from the TimeZoneNames strings.
1667         // Want to be able to parse both short and long forms.
1668         int zoneIndex = formatData.getZoneIndex(currentTimeZone.getID());
1669         TimeZone tz = null;
1670         String[] [] zoneStrings = formatData.getZoneStringsWrapper();
1671         String[] zoneNames = null;
1672         int nameIndex = 0;
1673         if (zoneIndex != -1) {
1674             zoneNames = zoneStrings[zoneIndex];
1675             if ((nameIndex = matchZoneString(text, start, zoneNames)) > 0) {
1676                 if (nameIndex <= 2) {
1677                     // Check if the standard name (abbr) and the daylight name are the same.
1678                     useSameName = zoneNames[nameIndex].equalsIgnoreCase(zoneNames[nameIndex + 2]);
1679                 }
1680                 tz = TimeZone.getTimeZone(zoneNames[0]);
1681             }
1682         }

```

```

1683     if (tz == null) {
1684         zoneIndex = formatData.getZoneIndex(TimeZone.getDefault().getID());
1685         if (zoneIndex != -1) {
1686             zoneNames = zoneStrings[zoneIndex];
1687             if ((nameIndex = matchZoneString(text, start, zoneNames)) > 0) {
1688                 if (nameIndex <= 2) {
1689                     useSameName = zoneNames[nameIndex].equalsIgnoreCase(zoneNames[nameIndex +
1690                                     2]);
1691                 }
1692                 tz = TimeZone.getTimeZone(zoneNames[0]);
1693             }
1694         }
1695     }
1696     if (tz == null) {
1697         int len = zoneStrings.length;
1698         for (int i = 0; i < len; i++) {
1699             zoneNames = zoneStrings[i];
1700             if ((nameIndex = matchZoneString(text, start, zoneNames)) > 0) {
1701                 if (nameIndex <= 2) {
1702                     useSameName = zoneNames[nameIndex].equalsIgnoreCase(zoneNames[nameIndex +
1703                                     2]);
1704                 }
1705                 tz = TimeZone.getTimeZone(zoneNames[0]);
1706                 break;
1707             }
1708         }
1709     }
1710     if (tz != null) { // Matched any ?
1711         if (!tz.equals(currentTimeZone)) {
1712             setTimeZone(tz);
1713         }
1714         // If the time zone matched uses the same name
1715         // (abbreviation) for both standard and daylight time,
1716         // let the time zone in the Calendar decide which one.
1717         //
1718         // Also if tz.getDSTSavings() returns 0 for DST, use tz to
1719         // determine the local time. (6645292)
1720         int dstAmount = (nameIndex >= 3) ? tz.getDSTSavings() : 0;
1721         if (!(useSameName || (nameIndex >= 3 && dstAmount == 0))) {
1722             calb.clear(Calendar.ZONE_OFFSET).set(Calendar.DST_OFFSET, dstAmount);
1723         }
1724         return (start + zoneNames[nameIndex].length());
1725     }
1726     return -start;
1727 }
1728 /**
1729  * Parses numeric forms of time zone offset, such as "hh:mm", and
1730  * sets calb to the parsed value.
1731  *
1732  * @param text the text to be parsed
1733  * @param start the character position to start parsing

```

```
1734 * @param sign 1: positive; -1: negative
1735 * @param count 0: 'Z' or "GMT+hh:mm" parsing; 1 - 3: the number of 'X's
1736 * @param colon true - colon required between hh and mm; false - no colon required
1737 * @param calb a CalendarBuilder in which the parsed value is stored
1738 * @return updated parsed position, or its negative value to indicate a parsing error
1739 */
1740 private int subParseNumericZone(String text, int start, int sign, int count,
1741                               boolean colon, CalendarBuilder calb) {
1742     int index = start;
1743
1744     parse:
1745     try {
1746         char c = text.charAt(index++);
1747         // Parse hh
1748         int hours;
1749         if (!isDigit(c)) {
1750             break parse;
1751         }
1752         hours = c - '0';
1753         c = text.charAt(index++);
1754         if (isDigit(c)) {
1755             hours = hours * 10 + (c - '0');
1756         } else {
1757             // If no colon in RFC 822 or 'X' (ISO), two digits are
1758             // required.
1759             if (count > 0 || !colon) {
1760                 break parse;
1761             }
1762             --index;
1763         }
1764         if (hours > 23) {
1765             break parse;
1766         }
1767         int minutes = 0;
1768         if (count != 1) {
1769             // Proceed with parsing mm
1770             c = text.charAt(index++);
1771             if (colon) {
1772                 if (c != ':') {
1773                     break parse;
1774                 }
1775                 c = text.charAt(index++);
1776             }
1777             if (!isDigit(c)) {
1778                 break parse;
1779             }
1780             minutes = c - '0';
1781             c = text.charAt(index++);
1782             if (!isDigit(c)) {
1783                 break parse;
1784             }
1785             minutes = minutes * 10 + (c - '0');
1786             if (minutes > 59) {
```

```

1787         break parse;
1788     }
1789 }
1790 minutes += hours * 60;
1791 calb.set(Calendar.ZONE_OFFSET, minutes * MILLIS_PER_MINUTE * sign)
1792     .set(Calendar.DST_OFFSET, 0);
1793     return index;
1794 } catch (IndexOutOfBoundsException e) {
1795 }
1796 return 1 - index; // -(index - 1)
1797 }
1798
1799 private boolean isDigit(char c) {
1800     return c >= '0' && c <= '9';
1801 }
1802
1803 /**
1804  * Private member function that converts the parsed date strings into
1805  * timeFields. Returns -start (for ParsePosition) if failed.
1806  * @param text the time text to be parsed.
1807  * @param start where to start parsing.
1808  * @param patternCharIndex the index of the pattern character.
1809  * @param count the count of a pattern character.
1810  * @param obeyCount if true, then the next field directly abuts this one,
1811  * and we should use the count to know when to stop parsing.
1812  * @param ambiguousYear return parameter; upon return, if ambiguousYear[0]
1813  * is true, then a two-digit year was parsed and may need to be readjusted.
1814  * @param origPos origPos.errorIndex is used to return an error index
1815  * at which a parse error occurred, if matching failure occurs.
1816  * @return the new start position if matching succeeded; -1 indicating
1817  * matching failure, otherwise. In case matching failure occurred,
1818  * an error index is set to origPos.errorIndex.
1819  */
1820 private int subParse(String text, int start, int patternCharIndex, int count,
1821     boolean obeyCount, boolean[] ambiguousYear,
1822     ParsePosition origPos,
1823     boolean useFollowingMinusSignAsDelimiter, CalendarBuilder calb) {
1824     Number number;
1825     int value = 0;
1826     ParsePosition pos = new ParsePosition(0);
1827     pos.index = start;
1828     if (patternCharIndex == PATTERN_WEEK_YEAR && !calendar.isWeekDateSupported()) {
1829         // use calendar year 'y' instead
1830         patternCharIndex = PATTERN_YEAR;
1831     }
1832     int field = PATTERN_INDEX_TO_CALENDAR_FIELD[patternCharIndex];
1833
1834     // If there are any spaces here, skip over them. If we hit the end
1835     // of the string, then fail.
1836     for (;;) {
1837         if (pos.index >= text.length()) {
1838             origPos.errorIndex = start;
1839             return -1;

```



```

1840     }
1841     char c = text.charAt(pos.index);
1842     if (c != ' ' && c != '\t') {
1843         break;
1844     }
1845     ++pos.index;
1846 }
1847 // Remember the actual start index
1848 int actualStart = pos.index;
1849
1850 parsing:
1851 {
1852     // We handle a few special cases here where we need to parse
1853     // a number value. We handle further, more generic cases below. We need
1854     // to handle some of them here because some fields require extra processing on
1855     // the parsed value.
1856     if (patternCharIndex == PATTERN_HOUR_OF_DAY1 ||
1857         patternCharIndex == PATTERN_HOUR1 ||
1858         (patternCharIndex == PATTERN_MONTH && count <= 2) ||
1859         patternCharIndex == PATTERN_YEAR ||
1860         patternCharIndex == PATTERN_WEEK_YEAR) {
1861         // It would be good to unify this with the obeyCount logic below,
1862         // but that's going to be difficult.
1863         if (obeyCount) {
1864             if ((start+count) > text.length()) {
1865                 break parsing;
1866             }
1867             number = numberFormat.parse(text.substring(0, start+count), pos);
1868         } else {
1869             number = numberFormat.parse(text, pos);
1870         }
1871         if (number == null) {
1872             if (patternCharIndex != PATTERN_YEAR || calendar instanceof GregorianCalendar)
1873             {
1874                 break parsing;
1875             }
1876         } else {
1877             value = number.intValue();
1878
1879             if (useFollowingMinusSignAsDelimiter && (value < 0) &&
1880                 (((pos.index < text.length()) &&
1881                  (text.charAt(pos.index) != minusSign)) ||
1882                 ((pos.index == text.length()) &&
1883                  (text.charAt(pos.index-1) == minusSign)))) {
1884                 value = -value;
1885                 pos.index--;
1886             }
1887         }
1888     }
1889     boolean useDateFormatSymbols = useDateFormatSymbols();
1890
1891     int index;

```

```

1892     switch (patternCharIndex) {
1893     case PATTERN_ERA: // 'G'
1894         if (useDateFormatSymbols) {
1895             if ((index = matchString(text, start, Calendar.ERA, formatData.getEras(), calb)
1896                 ) > 0) {
1897                 return index;
1898             }
1899         } else {
1900             Map<String, Integer> map = getDisplayNamesMap(field, locale);
1901             if ((index = matchString(text, start, field, map, calb)) > 0) {
1902                 return index;
1903             }
1904         }
1905         break parsing;
1906
1907     case PATTERN_WEEK_YEAR: // 'Y'
1908     case PATTERN_YEAR:      // 'y'
1909         if (!(calendar instanceof GregorianCalendar)) {
1910             // calendar might have text representations for year values,
1911             // such as "\u5143" in JapaneseImperialCalendar.
1912             int style = (count >= 4) ? Calendar.LONG : Calendar.SHORT;
1913             Map<String, Integer> map = calendar.getDisplayNames(field, style, locale);
1914             if (map != null) {
1915                 if ((index = matchString(text, start, field, map, calb)) > 0) {
1916                     return index;
1917                 }
1918             }
1919             calb.set(field, value);
1920             return pos.index;
1921         }
1922
1923         // If there are 3 or more YEAR pattern characters, this indicates
1924         // that the year value is to be treated literally, without any
1925         // two-digit year adjustments (e.g., from "01" to 2001). Otherwise
1926         // we made adjustments to place the 2-digit year in the proper
1927         // century, for parsed strings from "00" to "99". Any other string
1928         // is treated literally: "2250", "-1", "1", "002".
1929         if (count <= 2 && (pos.index - actualStart) == 2
1930             && Character.isDigit(text.charAt(actualStart))
1931             && Character.isDigit(text.charAt(actualStart + 1))) {
1932             // Assume for example that the defaultCenturyStart is 6/18/1903.
1933             // This means that two-digit years will be forced into the range
1934             // 6/18/1903 to 6/17/2003. As a result, years 00, 01, and 02
1935             // correspond to 2000, 2001, and 2002. Years 04, 05, etc. correspond
1936             // to 1904, 1905, etc. If the year is 03, then it is 2003 if the
1937             // other fields specify a date before 6/18, or 1903 if they specify a
1938             // date afterwards. As a result, 03 is an ambiguous year. All other
1939             // two-digit years are unambiguous.
1940             int ambiguousTwoDigitYear = defaultCenturyStartYear % 100;
1941             ambiguousYear[0] = value == ambiguousTwoDigitYear;
1942             value += (defaultCenturyStartYear/100)*100 +
1943                 (value < ambiguousTwoDigitYear ? 100 : 0);

```

```

1944         calb.set(field, value);
1945         return pos.index;
1946
1947     case PATTERN_MONTH: // 'M'
1948         if (count <= 2) // i.e., M or MM.
1949         {
1950             // Don't want to parse the month if it is a string
1951             // while pattern uses numeric style: M or MM.
1952             // [We computed 'value' above.]
1953             calb.set(Calendar.MONTH, value - 1);
1954             return pos.index;
1955         }
1956
1957         if (useDateFormatSymbols) {
1958             // count >= 3 // i.e., MMM or MMMM
1959             // Want to be able to parse both short and long forms.
1960             // Try count == 4 first:
1961             int newStart;
1962             if ((newStart = matchString(text, start, Calendar.MONTH,
1963                                     formatData.getMonths(), calb)) > 0) {
1964                 return newStart;
1965             }
1966             // count == 4 failed, now try count == 3
1967             if ((index = matchString(text, start, Calendar.MONTH,
1968                                     formatData.getShortMonths(), calb)) > 0) {
1969                 return index;
1970             }
1971         } else {
1972             Map<String, Integer> map = getDisplayNamesMap(field, locale);
1973             if ((index = matchString(text, start, field, map, calb)) > 0) {
1974                 return index;
1975             }
1976         }
1977         break parsing;
1978
1979     case PATTERN_HOUR_OF_DAY1: // 'k' 1-based.  eg, 23:59 + 1 hour ==> 24:59
1980         if (!isLenient()) {
1981             // Validate the hour value in non-lenient
1982             if (value < 1 || value > 24) {
1983                 break parsing;
1984             }
1985         }
1986         // [We computed 'value' above.]
1987         if (value == calendar.getMaximum(Calendar.HOUR_OF_DAY) + 1) {
1988             value = 0;
1989         }
1990         calb.set(Calendar.HOUR_OF_DAY, value);
1991         return pos.index;
1992
1993     case PATTERN_DAY_OF_WEEK: // 'E'
1994         {
1995             if (useDateFormatSymbols) {
1996                 // Want to be able to parse both short and long forms.

```

```

1997         // Try count == 4 (DDDD) first:
1998         int newStart;
1999         if ((newStart=matchString(text, start, Calendar.DAY_OF_WEEK,
2000             formatData.getWeekdays(), calb)) > 0) {
2001             return newStart;
2002         }
2003         // DDDD failed, now try DDD
2004         if ((index = matchString(text, start, Calendar.DAY_OF_WEEK,
2005             formatData.getShortWeekdays(), calb)) > 0) {
2006             return index;
2007         }
2008     } else {
2009         int[] styles = { Calendar.LONG, Calendar.SHORT };
2010         for (int style : styles) {
2011             Map<String,Integer> map = calendar.getDisplayNames(field, style, locale
2012                 );
2013             if ((index = matchString(text, start, field, map, calb)) > 0) {
2014                 return index;
2015             }
2016         }
2017     }
2018     break parsing;
2019
2020 case PATTERN_AM_PM:    // 'a'
2021     if (useDateFormatSymbols) {
2022         if ((index = matchString(text, start, Calendar.AM_PM,
2023             formatData.getAmPmStrings(), calb)) > 0) {
2024             return index;
2025         }
2026     } else {
2027         Map<String,Integer> map = getDisplayNamesMap(field, locale);
2028         if ((index = matchString(text, start, field, map, calb)) > 0) {
2029             return index;
2030         }
2031     }
2032     break parsing;
2033
2034 case PATTERN_HOUR1: // 'h' 1-based.  eg, 11PM + 1 hour ==> 12 AM
2035     if (!isLenient()) {
2036         // Validate the hour value in non-lenient
2037         if (value < 1 || value > 12) {
2038             break parsing;
2039         }
2040     }
2041     // [We computed 'value' above.]
2042     if (value == calendar.getLeastMaximum(Calendar.HOUR) + 1) {
2043         value = 0;
2044     }
2045     calb.set(Calendar.HOUR, value);
2046     return pos.index;
2047
2048 case PATTERN_ZONE_NAME: // 'z'

```

```
2049     case PATTERN_ZONE_VALUE: // 'Z'
2050     {
2051         int sign = 0;
2052         try {
2053             char c = text.charAt(pos.index);
2054             if (c == '+') {
2055                 sign = 1;
2056             } else if (c == '-') {
2057                 sign = -1;
2058             }
2059             if (sign == 0) {
2060                 // Try parsing a custom time zone "GMT+hh:mm" or "GMT".
2061                 if ((c == 'G' || c == 'g')
2062                     && (text.length() - start) >= GMT.length()
2063                     && text.regionMatches(true, start, GMT, 0, GMT.length())) {
2064                     pos.index = start + GMT.length();
2065
2066                     if ((text.length() - pos.index) > 0) {
2067                         c = text.charAt(pos.index);
2068                         if (c == '+') {
2069                             sign = 1;
2070                         } else if (c == '-') {
2071                             sign = -1;
2072                         }
2073                     }
2074
2075                     if (sign == 0) { /* "GMT" without offset */
2076                         calb.set(Calendar.ZONE_OFFSET, 0)
2077                             .set(Calendar.DST_OFFSET, 0);
2078                         return pos.index;
2079                     }
2080
2081                     // Parse the rest as "hh:mm"
2082                     int i = subParseNumericZone(text, ++pos.index,
2083                                                 sign, 0, true, calb);
2084                     if (i > 0) {
2085                         return i;
2086                     }
2087                     pos.index = -i;
2088                 } else {
2089                     // Try parsing the text as a time zone
2090                     // name or abbreviation.
2091                     int i = subParseZoneString(text, pos.index, calb);
2092                     if (i > 0) {
2093                         return i;
2094                     }
2095                     pos.index = -i;
2096                 }
2097             } else {
2098                 // Parse the rest as "hhmm" (RFC 822)
2099                 int i = subParseNumericZone(text, ++pos.index,
2100                                             sign, 0, false, calb);
2101                 if (i > 0) {
```

```

2102         return i;
2103     }
2104     pos.index = -i;
2105 }
2106 } catch (IndexOutOfBoundsException e) {
2107 }
2108 }
2109 break parsing;
2110
2111 case PATTERN_ISO_ZONE: // 'X'
2112 {
2113     if ((text.length() - pos.index) <= 0) {
2114         break parsing;
2115     }
2116
2117     int sign;
2118     char c = text.charAt(pos.index);
2119     if (c == 'Z') {
2120         calb.set(Calendar.ZONE_OFFSET, 0).set(Calendar.DST_OFFSET, 0);
2121         return ++pos.index;
2122     }
2123
2124     // parse text as "+/-hh[:mm]" based on count
2125     if (c == '+') {
2126         sign = 1;
2127     } else if (c == '-') {
2128         sign = -1;
2129     } else {
2130         ++pos.index;
2131         break parsing;
2132     }
2133     int i = subParseNumericZone(text, ++pos.index, sign, count,
2134                                count == 3, calb);
2135     if (i > 0) {
2136         return i;
2137     }
2138     pos.index = -i;
2139 }
2140 break parsing;
2141
2142 default:
2143 // case PATTERN_DAY_OF_MONTH: // 'd'
2144 // case PATTERN_HOUR_OF_DAY: // 'H' 0-based. eg, 23:59 + 1 hour ==> 00:59
2145 // case PATTERN_MINUTE: // 'm'
2146 // case PATTERN_SECOND: // 's'
2147 // case PATTERN_MILLISECOND: // 'S'
2148 // case PATTERN_DAY_OF_YEAR: // 'D'
2149 // case PATTERN_DAY_OF_WEEK_IN_MONTH: // 'F'
2150 // case PATTERN_WEEK_OF_YEAR: // 'w'
2151 // case PATTERN_WEEK_OF_MONTH: // 'W'
2152 // case PATTERN_HOURO: // 'K' 0-based. eg, 11PM + 1 hour ==> 0 AM
2153 // case PATTERN_ISO_DAY_OF_WEEK: // 'u' (pseudo field);
2154

```

```

2155         // Handle "generic" fields
2156         if (obeyCount) {
2157             if ((start+count) > text.length()) {
2158                 break parsing;
2159             }
2160             number = numberFormat.parse(text.substring(0, start+count), pos);
2161         } else {
2162             number = numberFormat.parse(text, pos);
2163         }
2164         if (number != null) {
2165             value = number.intValue();
2166
2167             if (useFollowingMinusSignAsDelimiter && (value < 0) &&
2168                 ((pos.index < text.length()) &&
2169                  (text.charAt(pos.index) != minusSign) ||
2170                  ((pos.index == text.length()) &&
2171                   (text.charAt(pos.index-1) == minusSign)))) {
2172                 value = -value;
2173                 pos.index--;
2174             }
2175
2176             calb.set(field, value);
2177             return pos.index;
2178         }
2179         break parsing;
2180     }
2181 }
2182
2183 // Parsing failed.
2184 origPos.errorIndex = pos.index;
2185 return -1;
2186 }
2187
2188 /**
2189  * Returns true if the DateFormatSymbols has been set explicitly or locale
2190  * is null.
2191  */
2192 private boolean useDateFormatSymbols() {
2193     return useDateFormatSymbols || locale == null;
2194 }
2195
2196 /**
2197  * Translates a pattern, mapping each character in the from string to the
2198  * corresponding character in the to string.
2199  *
2200  * @exception IllegalArgumentException if the given pattern is invalid
2201  */
2202 private String translatePattern(String pattern, String from, String to) {
2203     StringBuilder result = new StringBuilder();
2204     boolean inQuote = false;
2205     for (int i = 0; i < pattern.length(); ++i) {
2206         char c = pattern.charAt(i);
2207         if (inQuote) {

```

```

2208         if (c == '\\') {
2209             inQuote = false;
2210         }
2211     }
2212     else {
2213         if (c == '\\') {
2214             inQuote = true;
2215         } else if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')) {
2216             int ci = from.indexOf(c);
2217             if (ci >= 0) {
2218                 // patternChars is longer than localPatternChars due
2219                 // to serialization compatibility. The pattern letters
2220                 // unsupported by localPatternChars pass through.
2221                 if (ci < to.length()) {
2222                     c = to.charAt(ci);
2223                 }
2224             } else {
2225                 throw new IllegalArgumentException("Illegal pattern " +
2226                     " character '" +
2227                     c + "'");
2228             }
2229         }
2230     }
2231     result.append(c);
2232 }
2233 if (inQuote) {
2234     throw new IllegalArgumentException("Unfinished quote in pattern");
2235 }
2236 return result.toString();
2237 }
2238
2239 /**
2240  * Returns a pattern string describing this date format.
2241  *
2242  * @return a pattern string describing this date format.
2243  */
2244 public String toPattern() {
2245     return pattern;
2246 }
2247
2248 /**
2249  * Returns a localized pattern string describing this date format.
2250  *
2251  * @return a localized pattern string describing this date format.
2252  */
2253 public String toLocalizedPattern() {
2254     return translatePattern(pattern,
2255         DateFormatSymbols.patternChars,
2256         formatData.getLocalPatternChars());
2257 }
2258
2259 /**
2260  * Applies the given pattern string to this date format.

```



```

2261     *
2262     * @param pattern the new date and time pattern for this date format
2263     * @exception NullPointerException if the given pattern is null
2264     * @exception IllegalArgumentException if the given pattern is invalid
2265     */
2266     public void applyPattern(String pattern)
2267     {
2268         applyPatternImpl(pattern);
2269     }
2270
2271     private void applyPatternImpl(String pattern) {
2272         compiledPattern = compile(pattern);
2273         this.pattern = pattern;
2274     }
2275
2276     /**
2277     * Applies the given localized pattern string to this date format.
2278     *
2279     * @param pattern a String to be mapped to the new date and time format
2280     *         pattern for this format
2281     * @exception NullPointerException if the given pattern is null
2282     * @exception IllegalArgumentException if the given pattern is invalid
2283     */
2284     public void applyLocalizedPattern(String pattern) {
2285         String p = translatePattern(pattern,
2286                                     formatData.getLocalPatternChars(),
2287                                     DateFormatSymbols.patternChars);
2288         compiledPattern = compile(p);
2289         this.pattern = p;
2290     }
2291
2292     /**
2293     * Gets a copy of the date and time format symbols of this date format.
2294     *
2295     * @return the date and time format symbols of this date format
2296     * @see #setDateFormatSymbols
2297     */
2298     public DateFormatSymbols getDateFormatSymbols()
2299     {
2300         return (DateFormatSymbols)formatData.clone();
2301     }
2302
2303     /**
2304     * Sets the date and time format symbols of this date format.
2305     *
2306     * @param newFormatSymbols the new date and time format symbols
2307     * @exception NullPointerException if the given newFormatSymbols is null
2308     * @see #getDateFormatSymbols
2309     */
2310     public void setDateFormatSymbols(DateFormatSymbols newFormatSymbols)
2311     {
2312         this.formatData = (DateFormatSymbols)newFormatSymbols.clone();
2313         useDateFormatSymbols = true;

```

```

2314     }
2315
2316     /**
2317      * Creates a copy of this SimpleDateFormat. This also
2318      * clones the format's date format symbols.
2319      *
2320      * @return a clone of this SimpleDateFormat
2321      */
2322     @Override
2323     public Object clone() {
2324         SimpleDateFormat other = (SimpleDateFormat) super.clone();
2325         other.formatData = (DateFormatSymbols) formatData.clone();
2326         return other;
2327     }
2328
2329     /**
2330      * Returns the hash code value for this SimpleDateFormat object.
2331      *
2332      * @return the hash code value for this SimpleDateFormat object.
2333      */
2334     @Override
2335     public int hashCode()
2336     {
2337         return pattern.hashCode();
2338         // just enough fields for a reasonable distribution
2339     }
2340
2341     /**
2342      * Compares the given object with this SimpleDateFormat for
2343      * equality.
2344      *
2345      * @return true if the given object is equal to this
2346      *         SimpleDateFormat
2347      */
2348     @Override
2349     public boolean equals(Object obj)
2350     {
2351         if (!super.equals(obj)) {
2352             return false; // super does class check
2353         }
2354         SimpleDateFormat that = (SimpleDateFormat) obj;
2355         return (pattern.equals(that.pattern)
2356             && formatData.equals(that.formatData));
2357     }
2358
2359     private static final int[] REST_OF_STYLES = {
2360         Calendar.SHORT_STANDALONE, Calendar.LONG_FORMAT, Calendar.LONG_STANDALONE,
2361     };
2362     private Map<String, Integer> getDisplayNamesMap(int field, Locale locale) {
2363         Map<String, Integer> map = calendar.getDisplayNames(field, Calendar.SHORT_FORMAT, locale);
2364         // Get all SHORT and LONG styles (avoid NARROW styles).
2365         for (int style : REST_OF_STYLES) {
2366             Map<String, Integer> m = calendar.getDisplayNames(field, style, locale);

```

```

2367         if (m != null) {
2368             map.putAll(m);
2369         }
2370     }
2371     return map;
2372 }
2373
2374 /**
2375  * After reading an object from the input stream, the format
2376  * pattern in the object is verified.
2377  * <p>
2378  * @exception InvalidObjectException if the pattern is invalid
2379  */
2380 private void readObject(ObjectInputStream stream)
2381     throws IOException, ClassNotFoundException {
2382     stream.defaultReadObject();
2383
2384     try {
2385         compiledPattern = compile(pattern);
2386     } catch (Exception e) {
2387         throw new InvalidObjectException("invalid pattern");
2388     }
2389
2390     if (serialVersionOnStream < 1) {
2391         // didn't have defaultCenturyStart field
2392         initializeDefaultCentury();
2393     }
2394     else {
2395         // fill in dependent transient field
2396         parseAmbiguousDatesAsAfter(defaultCenturyStart);
2397     }
2398     serialVersionOnStream = currentSerialVersion;
2399
2400     // If the deserialized object has a SimpleTimeZone, try
2401     // to replace it with a ZoneInfo equivalent in order to
2402     // be compatible with the SimpleTimeZone-based
2403     // implementation as much as possible.
2404     TimeZone tz = getTimeZone();
2405     if (tz instanceof SimpleTimeZone) {
2406         String id = tz.getID();
2407         TimeZone zi = TimeZone.getTimeZone(id);
2408         if (zi != null && zi.hasSameRules(tz) && zi.getID().equals(id)) {
2409             setTimeZone(zi);
2410         }
2411     }
2412 }
2413
2414 /**
2415  * Analyze the negative subpattern of DecimalFormat and set/update values
2416  * as necessary.
2417  */
2418 private void checkNegativeNumberExpression() {
2419     if ((numberFormat instanceof DecimalFormat) &&

```

```

2420         !numberFormat.equals(originalNumberFormat)) {
2421         String numberPattern = ((DecimalFormat)numberFormat).toPattern();
2422         if (!numberPattern.equals(originalNumberPattern)) {
2423             hasFollowingMinusSign = false;
2424
2425             int separatorIndex = numberPattern.indexOf(';');
2426             // If the negative subpattern is not absent, we have to analyze
2427             // it in order to check if it has a following minus sign.
2428             if (separatorIndex > -1) {
2429                 int minusIndex = numberPattern.indexOf('-', separatorIndex);
2430                 if ((minusIndex > numberPattern.lastIndexOf('0')) &&
2431                     (minusIndex > numberPattern.lastIndexOf('#'))) {
2432                     hasFollowingMinusSign = true;
2433                     minusSign = ((DecimalFormat)numberFormat).getDecimalFormatSymbols().
                        getMinusSign();
2434                 }
2435             }
2436             originalNumberPattern = numberPattern;
2437         }
2438         originalNumberFormat = numberFormat;
2439     }
2440 }
2441
2442 }

```

2.34 StringCharacterIterator.java

Secuencia 43: java/text/StringCharacterIterator.java

```

1  /*
2  * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25

```

```
26  /*
27  * (C) Copyright Taligent, Inc. 1996, 1997 - All Rights Reserved
28  * (C) Copyright IBM Corp. 1996 - 1998 - All Rights Reserved
29  *
30  * The original version of this source code and documentation
31  * is copyrighted and owned by Taligent, Inc., a wholly-owned
32  * subsidiary of IBM. These materials are provided under terms
33  * of a License Agreement between Taligent and Sun. This technology
34  * is protected by multiple US and International patents.
35  *
36  * This notice and attribution to Taligent may not be removed.
37  * Taligent is a registered trademark of Taligent, Inc.
38  *
39  */
40
41  package java.text;
42
43  /**
44   * <code>StringCharacterIterator</code> implements the
45   * <code>CharacterIterator</code> protocol for a <code>String</code>.
46   * The <code>StringCharacterIterator</code> class iterates over the
47   * entire <code>String</code>.
48   *
49   * @see CharacterIterator
50   */
51
52  public final class StringCharacterIterator implements CharacterIterator
53  {
54      private String text;
55      private int begin;
56      private int end;
57      // invariant: begin <= pos <= end
58      private int pos;
59
60      /**
61       * Constructs an iterator with an initial index of 0.
62       *
63       * @param text the {@code String} to be iterated over
64       */
65      public StringCharacterIterator(String text)
66      {
67          this(text, 0);
68      }
69
70      /**
71       * Constructs an iterator with the specified initial index.
72       *
73       * @param text  The String to be iterated over
74       * @param pos    Initial iterator position
75       */
76      public StringCharacterIterator(String text, int pos)
77      {
78          this(text, 0, text.length(), pos);
```

```
79     }
80
81     /**
82      * Constructs an iterator over the given range of the given string, with the
83      * index set at the specified position.
84      *
85      * @param text    The String to be iterated over
86      * @param begin    Index of the first character
87      * @param end      Index of the character following the last character
88      * @param pos      Initial iterator position
89      */
90     public StringCharacterIterator(String text, int begin, int end, int pos) {
91         if (text == null)
92             throw new NullPointerException();
93         this.text = text;
94
95         if (begin < 0 || begin > end || end > text.length())
96             throw new IllegalArgumentException("Invalid substring range");
97
98         if (pos < begin || pos > end)
99             throw new IllegalArgumentException("Invalid position");
100
101         this.begin = begin;
102         this.end = end;
103         this.pos = pos;
104     }
105
106     /**
107      * Reset this iterator to point to a new string. This package-visible
108      * method is used by other java.text classes that want to avoid allocating
109      * new StringCharacterIterator objects every time their setText method
110      * is called.
111      *
112      * @param text    The String to be iterated over
113      * @since 1.2
114      */
115     public void setText(String text) {
116         if (text == null)
117             throw new NullPointerException();
118         this.text = text;
119         this.begin = 0;
120         this.end = text.length();
121         this.pos = 0;
122     }
123
124     /**
125      * Implements CharacterIterator.first() for String.
126      * @see CharacterIterator#first
127      */
128     public char first()
129     {
130         pos = begin;
131         return current();
132     }
```

```
132     }
133
134     /**
135      * Implements CharacterIterator.last() for String.
136      * @see CharacterIterator#last
137      */
138     public char last()
139     {
140         if (end != begin) {
141             pos = end - 1;
142         } else {
143             pos = end;
144         }
145         return current();
146     }
147
148     /**
149      * Implements CharacterIterator.setIndex() for String.
150      * @see CharacterIterator#setIndex
151      */
152     public char setIndex(int p)
153     {
154         if (p < begin || p > end)
155             throw new IllegalArgumentException("Invalid index");
156         pos = p;
157         return current();
158     }
159
160     /**
161      * Implements CharacterIterator.current() for String.
162      * @see CharacterIterator#current
163      */
164     public char current()
165     {
166         if (pos >= begin && pos < end) {
167             return text.charAt(pos);
168         }
169         else {
170             return DONE;
171         }
172     }
173
174     /**
175      * Implements CharacterIterator.next() for String.
176      * @see CharacterIterator#next
177      */
178     public char next()
179     {
180         if (pos < end - 1) {
181             pos++;
182             return text.charAt(pos);
183         }
184         else {
```

```
185         pos = end;
186         return DONE;
187     }
188 }
189
190 /**
191  * Implements CharacterIterator.previous() for String.
192  * @see CharacterIterator#previous
193  */
194 public char previous()
195 {
196     if (pos > begin) {
197         pos--;
198         return text.charAt(pos);
199     }
200     else {
201         return DONE;
202     }
203 }
204
205 /**
206  * Implements CharacterIterator.getBeginIndex() for String.
207  * @see CharacterIterator#getBeginIndex
208  */
209 public int getBeginIndex()
210 {
211     return begin;
212 }
213
214 /**
215  * Implements CharacterIterator.getEndIndex() for String.
216  * @see CharacterIterator#getEndIndex
217  */
218 public int getEndIndex()
219 {
220     return end;
221 }
222
223 /**
224  * Implements CharacterIterator.getIndex() for String.
225  * @see CharacterIterator#getIndex
226  */
227 public int getIndex()
228 {
229     return pos;
230 }
231
232 /**
233  * Compares the equality of two StringCharacterIterator objects.
234  * @param obj the StringCharacterIterator object to be compared with.
235  * @return true if the given obj is the same as this
236  *         StringCharacterIterator object; false otherwise.
237  */
```



```

238 public boolean equals(Object obj)
239 {
240     if (this == obj)
241         return true;
242     if (!(obj instanceof StringCharacterIterator))
243         return false;
244
245     StringCharacterIterator that = (StringCharacterIterator) obj;
246
247     if (hashCode() != that.hashCode())
248         return false;
249     if (!text.equals(that.text))
250         return false;
251     if (pos != that.pos || begin != that.begin || end != that.end)
252         return false;
253     return true;
254 }
255
256 /**
257  * Computes a hashcode for this iterator.
258  * @return A hash code
259  */
260 public int hashCode()
261 {
262     return text.hashCode() ^ pos ^ begin ^ end;
263 }
264
265 /**
266  * Creates a copy of this iterator.
267  * @return A copy of this
268  */
269 public Object clone()
270 {
271     try {
272         StringCharacterIterator other
273             = (StringCharacterIterator) super.clone();
274         return other;
275     }
276     catch (CloneNotSupportedException e) {
277         throw new InternalError(e);
278     }
279 }
280
281 }

```

3 Text Spi (java.text.spi package)

3.1 BreakIteratorProvider.java

Secuencia 44: java/text/spi/BreakIteratorProvider.java

```

1  /*
2  * Copyright (c) 2005, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.

```

```
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 package java.text.spi;
27
28 import java.text.BreakIterator;
29 import java.util.Locale;
30 import java.util.spi.LocaleServiceProvider;
31
32 /**
33  * An abstract class for service providers that
34  * provide concrete implementations of the
35  * {@link java.text.BreakIterator BreakIterator} class.
36  *
37  * @since      1.6
38  */
39 public abstract class BreakIteratorProvider extends LocaleServiceProvider {
40
41     /**
42      * Sole constructor. (For invocation by subclass constructors, typically
43      * implicit.)
44      */
45     protected BreakIteratorProvider() {
46     }
47
48     /**
49      * Returns a new BreakIterator instance
50      * for word breaks
51      * for the given locale.
52      * @param locale the desired locale
53      * @return A break iterator for word breaks
54      * @exception NullPointerException if locale is null
55      * @exception IllegalArgumentException if locale isn't
56      *         one of the locales returned from
```

```

57     *      {@link java.util.spi.LocaleServiceProvider#getAvailableLocales()
58     *      getAvailableLocales()}.
59     * @see java.text.BreakIterator#getWordInstance(java.util.Locale)
60     */
61     public abstract BreakIterator getWordInstance(Locale locale);
62
63     /**
64     * Returns a new BreakIterator instance
65     * for line breaks
66     * for the given locale.
67     * @param locale the desired locale
68     * @return A break iterator for line breaks
69     * @exception NullPointerException if locale is null
70     * @exception IllegalArgumentException if locale isn't
71     *         one of the locales returned from
72     *         {@link java.util.spi.LocaleServiceProvider#getAvailableLocales()
73     *         getAvailableLocales()}.
74     * @see java.text.BreakIterator#getLineInstance(java.util.Locale)
75     */
76     public abstract BreakIterator getLineInstance(Locale locale);
77
78     /**
79     * Returns a new BreakIterator instance
80     * for character breaks
81     * for the given locale.
82     * @param locale the desired locale
83     * @return A break iterator for character breaks
84     * @exception NullPointerException if locale is null
85     * @exception IllegalArgumentException if locale isn't
86     *         one of the locales returned from
87     *         {@link java.util.spi.LocaleServiceProvider#getAvailableLocales()
88     *         getAvailableLocales()}.
89     * @see java.text.BreakIterator#getCharacterInstance(java.util.Locale)
90     */
91     public abstract BreakIterator getCharacterInstance(Locale locale);
92
93     /**
94     * Returns a new BreakIterator instance
95     * for sentence breaks
96     * for the given locale.
97     * @param locale the desired locale
98     * @return A break iterator for sentence breaks
99     * @exception NullPointerException if locale is null
100    * @exception IllegalArgumentException if locale isn't
101    *         one of the locales returned from
102    *         {@link java.util.spi.LocaleServiceProvider#getAvailableLocales()
103    *         getAvailableLocales()}.
104    * @see java.text.BreakIterator#getSentenceInstance(java.util.Locale)
105    */
106    public abstract BreakIterator getSentenceInstance(Locale locale);
107 }

```

3.2 CollatorProvider.java

Secuencia 45: java/text/spi/CollatorProvider.java

```
1  /*
2  * Copyright (c) 2005, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 package java.text.spi;
27
28 import java.text.Collator;
29 import java.util.Locale;
30 import java.util.spi.LocaleServiceProvider;
31
32 /**
33  * An abstract class for service providers that
34  * provide concrete implementations of the
35  * {@link java.text.Collator Collator} class.
36  *
37  * @since      1.6
38  */
39 public abstract class CollatorProvider extends LocaleServiceProvider {
40
41     /**
42      * Sole constructor. (For invocation by subclass constructors, typically
43      * implicit.)
44      */
45     protected CollatorProvider() {
46     }
47
48     /**
49      * Returns a new <code>Collator</code> instance for the specified locale.
50      * @param locale the desired locale.
```

```

51     * @return the <code>Collator</code> for the desired locale.
52     * @exception NullPointerException if
53     * <code>locale</code> is null
54     * @exception IllegalArgumentException if <code>locale</code> isn't
55     *     one of the locales returned from
56     *     {@link java.util.spi.LocaleServiceProvider#getAvailableLocales()
57     *     getAvailableLocales()}.
58     * @see java.text.Collator#getInstance(java.util.Locale)
59     */
60     public abstract Collator getInstance(Locale locale);
61 }

```

3.3 DateFormatProvider.java

Secuencia 46: java/text/spi/DateFormatProvider.java

```

1  /*
2  * Copyright (c) 2005, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 package java.text.spi;
27
28 import java.text.DateFormat;
29 import java.util.Locale;
30 import java.util.spi.LocaleServiceProvider;
31
32 /**
33  * An abstract class for service providers that
34  * provide concrete implementations of the
35  * {@link java.text.DateFormat DateFormat} class.
36  *
37  * @since      1.6
38  */

```

```
39 public abstract class DateFormatProvider extends LocaleServiceProvider {
40
41     /**
42      * Sole constructor. (For invocation by subclass constructors, typically
43      * implicit.)
44      */
45     protected DateFormatProvider() {
46     }
47
48     /**
49      * Returns a new DateFormat instance which formats time
50      * with the given formatting style for the specified locale.
51      * @param style the given formatting style. Either one of
52      *     {@link java.text.DateFormat#SHORT DateFormat.SHORT},
53      *     {@link java.text.DateFormat#MEDIUM DateFormat.MEDIUM},
54      *     {@link java.text.DateFormat#LONG DateFormat.LONG}, or
55      *     {@link java.text.DateFormat#FULL DateFormat.FULL}.
56      * @param locale the desired locale.
57      * @exception IllegalArgumentException if style is invalid,
58      *     or if locale isn't
59      *     one of the locales returned from
60      *     {@link java.util.spi.LocaleServiceProvider#getAvailableLocales()
61      *     getAvailableLocales()}.
62      * @exception NullPointerException if locale is null
63      * @return a time formatter.
64      * @see java.text.DateFormat#getTimeInstance(int, java.util.Locale)
65      */
66     public abstract DateFormat getTimeInstance(int style, Locale locale);
67
68     /**
69      * Returns a new DateFormat instance which formats date
70      * with the given formatting style for the specified locale.
71      * @param style the given formatting style. Either one of
72      *     {@link java.text.DateFormat#SHORT DateFormat.SHORT},
73      *     {@link java.text.DateFormat#MEDIUM DateFormat.MEDIUM},
74      *     {@link java.text.DateFormat#LONG DateFormat.LONG}, or
75      *     {@link java.text.DateFormat#FULL DateFormat.FULL}.
76      * @param locale the desired locale.
77      * @exception IllegalArgumentException if style is invalid,
78      *     or if locale isn't
79      *     one of the locales returned from
80      *     {@link java.util.spi.LocaleServiceProvider#getAvailableLocales()
81      *     getAvailableLocales()}.
82      * @exception NullPointerException if locale is null
83      * @return a date formatter.
84      * @see java.text.DateFormat#getDateInstance(int, java.util.Locale)
85      */
86     public abstract DateFormat getDateInstance(int style, Locale locale);
87
88     /**
89      * Returns a new DateFormat instance which formats date and time
90      * with the given formatting style for the specified locale.
91      * @param dateStyle the given date formatting style. Either one of
```

```

92     * {@link java.text.DateFormat#SHORT DateFormat.SHORT},
93     * {@link java.text.DateFormat#MEDIUM DateFormat.MEDIUM},
94     * {@link java.text.DateFormat#LONG DateFormat.LONG}, or
95     * {@link java.text.DateFormat#FULL DateFormat.FULL}.
96     * @param timeStyle the given time formatting style. Either one of
97     *     {@link java.text.DateFormat#SHORT DateFormat.SHORT},
98     *     {@link java.text.DateFormat#MEDIUM DateFormat.MEDIUM},
99     *     {@link java.text.DateFormat#LONG DateFormat.LONG}, or
100    *     {@link java.text.DateFormat#FULL DateFormat.FULL}.
101    * @param locale the desired locale.
102    * @exception IllegalArgumentException if <code>dateStyle</code> or
103    *     <code>timeStyle</code> is invalid,
104    *     or if <code>locale</code> isn't
105    *     one of the locales returned from
106    *     {@link java.util.spi.LocaleServiceProvider#getAvailableLocales()}
107    *     getAvailableLocales()}.
108    * @exception NullPointerException if <code>locale</code> is null
109    * @return a date/time formatter.
110    * @see java.text.DateFormat#getTimeInstance(int, int, java.util.Locale)
111    */
112    public abstract DateFormat
113        getTimeInstance(int dateStyle, int timeStyle, Locale locale);
114 }

```

3.4 DateFormatSymbolsProvider.java

Secuencia 47: java/text/spi/DateFormatSymbolsProvider.java

```

1  /*
2  * Copyright (c) 2005, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 package java.text.spi;

```

```

27
28 import java.text.DateFormatSymbols;
29 import java.util.Locale;
30 import java.util.spi.LocaleServiceProvider;
31
32 /**
33  * An abstract class for service providers that
34  * provide instances of the
35  * {@link java.text.DateFormatSymbols DateFormatSymbols} class.
36  *
37  * @since      1.6
38  */
39 public abstract class DateFormatSymbolsProvider extends LocaleServiceProvider {
40
41     /**
42      * Sole constructor. (For invocation by subclass constructors, typically
43      * implicit.)
44      */
45     protected DateFormatSymbolsProvider() {
46     }
47
48     /**
49      * Returns a new DateFormatSymbols instance for the
50      * specified locale.
51      *
52      * @param locale the desired locale
53      * @exception NullPointerException if locale is null
54      * @exception IllegalArgumentException if locale isn't
55      *         one of the locales returned from
56      *         {@link java.util.spi.LocaleServiceProvider#getAvailableLocales()
57      *         getAvailableLocales()}.
58      * @return a DateFormatSymbols instance.
59      * @see java.text.DateFormatSymbols#getInstance(java.util.Locale)
60      */
61     public abstract DateFormatSymbols getInstance(Locale locale);
62 }

```

3.5 DecimalFormatSymbolsProvider.java

Secuencia 48: java/text/spi/DecimalFormatSymbolsProvider.java

```

1  /*
2  * Copyright (c) 2005, 2012, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *

```



```

14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  package java.text.spi;
27
28  import java.text.DecimalFormatSymbols;
29  import java.util.Locale;
30  import java.util.spi.LocaleServiceProvider;
31
32  /**
33   * An abstract class for service providers that
34   * provide instances of the
35   * {@link java.text.DecimalFormatSymbols DecimalFormatSymbols} class.
36   *
37   * <p>The requested {@code Locale} may contain an <a
38   * href="../../util/Locale.html#def_locale_extension"> extension</a> for
39   * specifying the desired numbering system. For example, {@code "ar-u-nu-arab"}
40   * (in the BCP 47 language tag form) specifies Arabic with the Arabic-Indic
41   * digits and symbols, while {@code "ar-u-nu-latn"} specifies Arabic with the
42   * Latin digits and symbols. Refer to the <em>Unicode Locale Data Markup
43   * Language (LDML)</em> specification for numbering systems.
44   *
45   * @since      1.6
46   * @see Locale#forLanguageTag(String)
47   * @see Locale#getExtension(char)
48   */
49  public abstract class DecimalFormatSymbolsProvider extends LocaleServiceProvider {
50
51      /**
52       * Sole constructor. (For invocation by subclass constructors, typically
53       * implicit.)
54       */
55      protected DecimalFormatSymbolsProvider() {
56      }
57
58      /**
59       * Returns a new <code>DecimalFormatSymbols</code> instance for the
60       * specified locale.
61       *
62       * @param locale the desired locale
63       * @exception NullPointerException if <code>locale</code> is null
64       * @exception IllegalArgumentException if <code>locale</code> isn't
65       *         one of the locales returned from
66       *         {@link java.util.spi.LocaleServiceProvider#getAvailableLocales()}

```

```

67     *     getAvailableLocales()}.
68     * @return a <code>DecimalFormatSymbols</code> instance.
69     * @see java.text.DecimalFormatSymbols#getInstance(java.util.Locale)
70     */
71     public abstract DecimalFormatSymbols getInstance(Locale locale);
72 }

```

3.6 NumberFormatProvider.java

Secuencia 49: java/text/spi/NumberFormatProvider.java

```

1  /*
2  * Copyright (c) 2005, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 package java.text.spi;
27
28 import java.text.NumberFormat;
29 import java.util.Locale;
30 import java.util.spi.LocaleServiceProvider;
31
32 /**
33  * An abstract class for service providers that
34  * provide concrete implementations of the
35  * {@link java.text.NumberFormat NumberFormat} class.
36  *
37  * @since      1.6
38  */
39 public abstract class NumberFormatProvider extends LocaleServiceProvider {
40
41     /**
42      * Sole constructor. (For invocation by subclass constructors, typically
43      * implicit.)

```

```

44     */
45     protected NumberFormatProvider() {
46     }
47
48     /**
49     * Returns a new <code>NumberFormat</code> instance which formats
50     * monetary values for the specified locale.
51     *
52     * @param locale the desired locale.
53     * @exception NullPointerException if <code>locale</code> is null
54     * @exception IllegalArgumentException if <code>locale</code> isn't
55     *     one of the locales returned from
56     *     {@link java.util.spi.LocaleServiceProvider#getAvailableLocales()
57     *     getAvailableLocales()}.
58     * @return a currency formatter
59     * @see java.text.NumberFormat#getCurrencyInstance(java.util.Locale)
60     */
61     public abstract NumberFormat getCurrencyInstance(Locale locale);
62
63     /**
64     * Returns a new <code>NumberFormat</code> instance which formats
65     * integer values for the specified locale.
66     * The returned number format is configured to
67     * round floating point numbers to the nearest integer using
68     * half-even rounding (see {@link java.math.RoundingMode#HALF_EVEN HALF_EVEN})
69     * for formatting, and to parse only the integer part of
70     * an input string (see {@link
71     * java.text.NumberFormat#isParseIntegerOnly isParseIntegerOnly}).
72     *
73     * @param locale the desired locale
74     * @exception NullPointerException if <code>locale</code> is null
75     * @exception IllegalArgumentException if <code>locale</code> isn't
76     *     one of the locales returned from
77     *     {@link java.util.spi.LocaleServiceProvider#getAvailableLocales()
78     *     getAvailableLocales()}.
79     * @return a number format for integer values
80     * @see java.text.NumberFormat#getIntegerInstance(java.util.Locale)
81     */
82     public abstract NumberFormat getIntegerInstance(Locale locale);
83
84     /**
85     * Returns a new general-purpose <code>NumberFormat</code> instance for
86     * the specified locale.
87     *
88     * @param locale the desired locale
89     * @exception NullPointerException if <code>locale</code> is null
90     * @exception IllegalArgumentException if <code>locale</code> isn't
91     *     one of the locales returned from
92     *     {@link java.util.spi.LocaleServiceProvider#getAvailableLocales()
93     *     getAvailableLocales()}.
94     * @return a general-purpose number formatter
95     * @see java.text.NumberFormat#getNumberInstance(java.util.Locale)
96     */

```

```
97     public abstract NumberFormat getNumberInstance(Locale locale);
98
99     /**
100      * Returns a new NumberFormat instance which formats
101      * percentage values for the specified locale.
102      *
103      * @param locale the desired locale
104      * @exception NullPointerException if locale is null
105      * @exception IllegalArgumentException if locale isn't
106      *         one of the locales returned from
107      *         {@link java.util.spi.LocaleServiceProvider#getAvailableLocales()
108      *         getAvailableLocales()}.
109      * @return a percent formatter
110      * @see java.text.NumberFormat#getPercentInstance(java.util.Locale)
111      */
112     public abstract NumberFormat getPercentInstance(Locale locale);
113 }
```

4 Time (java.time package)

4.1 Clock.java

Secuencia 50: java/time/Clock.java

```
1  /*
2   * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3   * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4   *
5   *
6   *
7   *
8   *
9   *
10  *
11  *
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26 /*
27  *
28  *
29  *
30  *
```

```
31 *
32 * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
62 package java.time;
63
64 import static java.time.LocalTime.NANOS_PER_MINUTE;
65 import static java.time.LocalTime.NANOS_PER_SECOND;
66
67 import java.io.Serializable;
68 import java.util.Objects;
69 import java.util.TimeZone;
70
71 /**
72  * A clock providing access to the current instant, date and time using a time-zone.
73  * <p>
74  * Instances of this class are used to find the current instant, which can be
75  * interpreted using the stored time-zone to find the current date and time.
76  * As such, a clock can be used instead of {@link System#currentTimeMillis()}
77  * and {@link TimeZone#getDefault()}.
78  * <p>
79  * Use of a {@code Clock} is optional. All key date-time classes also have a
80  * {@code now()} factory method that uses the system clock in the default time zone.
81  * The primary purpose of this abstraction is to allow alternate clocks to be
82  * plugged in as and when required. Applications use an object to obtain the
83  * current time rather than a static method. This can simplify testing.
```

```

84 * <p>
85 * Best practice for applications is to pass a {@code Clock} into any method
86 * that requires the current instant. A dependency injection framework is one
87 * way to achieve this:
88 * <pre>
89 * public class MyBean {
90 *     private Clock clock; // dependency inject
91 *     ...
92 *     public void process(LocalDate eventDate) {
93 *         if (eventDate.isBefore(LocalDate.now(clock)) {
94 *             ...
95 *         }
96 *     }
97 * }
98 * </pre>
99 * This approach allows an alternate clock, such as {@link #fixed(Instant, ZoneId) fixed}
100 * or {@link #offset(Clock, Duration) offset} to be used during testing.
101 * <p>
102 * The {@code system} factory methods provide clocks based on the best available
103 * system clock This may use {@link System#currentTimeMillis()}, or a higher
104 * resolution clock if one is available.
105 *
106 * @implSpec
107 * This abstract class must be implemented with care to ensure other classes operate correctly.
108 * All implementations that can be instantiated must be final, immutable and thread-safe.
109 * <p>
110 * The principal methods are defined to allow the throwing of an exception.
111 * In normal use, no exceptions will be thrown, however one possible implementation would be to
112 * obtain the time from a central time server across the network. Obviously, in this case the
113 * lookup could fail, and so the method is permitted to throw an exception.
114 * <p>
115 * The returned instants from {@code Clock} work on a time-scale that ignores leap seconds,
116 * as described in {@link Instant}. If the implementation wraps a source that provides leap
117 * second information, then a mechanism should be used to "smooth" the leap second.
118 * The Java Time-Scale mandates the use of UTC-SLS, however clock implementations may choose
119 * how accurate they are with the time-scale so long as they document how they work.
120 * Implementations are therefore not required to actually perform the UTC-SLS slew or to
121 * otherwise be aware of leap seconds.
122 * <p>
123 * Implementations should implement {@code Serializable} wherever possible and must
124 * document whether or not they do support serialization.
125 *
126 * @implNote
127 * The clock implementation provided here is based on {@link System#currentTimeMillis()}.
128 * That method provides little to no guarantee about the accuracy of the clock.
129 * Applications requiring a more accurate clock must implement this abstract class
130 * themselves using a different external clock, such as an NTP server.
131 *
132 * @since 1.8
133 */
134 public abstract class Clock {
135
136     /**

```

```
137     * Obtains a clock that returns the current instant using the best available
138     * system clock, converting to date and time using the UTC time-zone.
139     * <p>
140     * This clock, rather than {@link #systemDefaultZone()}, should be used when
141     * you need the current instant without the date or time.
142     * <p>
143     * This clock is based on the best available system clock.
144     * This may use {@link System#currentTimeMillis()}, or a higher resolution
145     * clock if one is available.
146     * <p>
147     * Conversion from instant to date or time uses the {@linkplain ZoneOffset#UTC UTC time-zone}.
148     * <p>
149     * The returned implementation is immutable, thread-safe and {@code Serializable}.
150     * It is equivalent to {@code system(ZoneOffset.UTC)}.
151     *
152     * @return a clock that uses the best available system clock in the UTC zone, not null
153     */
154     public static Clock systemUTC() {
155         return new SystemClock(ZoneOffset.UTC);
156     }
157
158     /**
159     * Obtains a clock that returns the current instant using the best available
160     * system clock, converting to date and time using the default time-zone.
161     * <p>
162     * This clock is based on the best available system clock.
163     * This may use {@link System#currentTimeMillis()}, or a higher resolution
164     * clock if one is available.
165     * <p>
166     * Using this method hard codes a dependency to the default time-zone into your application.
167     * It is recommended to avoid this and use a specific time-zone whenever possible.
168     * The {@link #systemUTC() UTC clock} should be used when you need the current instant
169     * without the date or time.
170     * <p>
171     * The returned implementation is immutable, thread-safe and {@code Serializable}.
172     * It is equivalent to {@code system(ZoneId.systemDefault())}.
173     *
174     * @return a clock that uses the best available system clock in the default zone, not null
175     * @see ZoneId#systemDefault()
176     */
177     public static Clock systemDefaultZone() {
178         return new SystemClock(ZoneId.systemDefault());
179     }
180
181     /**
182     * Obtains a clock that returns the current instant using best available
183     * system clock.
184     * <p>
185     * This clock is based on the best available system clock.
186     * This may use {@link System#currentTimeMillis()}, or a higher resolution
187     * clock if one is available.
188     * <p>
189     * Conversion from instant to date or time uses the specified time-zone.
```

```

190     * <p>
191     * The returned implementation is immutable, thread-safe and {@code Serializable}.
192     *
193     * @param zone the time-zone to use to convert the instant to date-time, not null
194     * @return a clock that uses the best available system clock in the specified zone, not null
195     */
196     public static Clock system(ZoneId zone) {
197         Objects.requireNonNull(zone, "zone");
198         return new SystemClock(zone);
199     }
200
201     //-----
202     /**
203     * Obtains a clock that returns the current instant ticking in whole seconds
204     * using best available system clock.
205     * <p>
206     * This clock will always have the nano-of-second field set to zero.
207     * This ensures that the visible time ticks in whole seconds.
208     * The underlying clock is the best available system clock, equivalent to
209     * using {@link #system(ZoneId)}.
210     * <p>
211     * Implementations may use a caching strategy for performance reasons.
212     * As such, it is possible that the start of the second observed via this
213     * clock will be later than that observed directly via the underlying clock.
214     * <p>
215     * The returned implementation is immutable, thread-safe and {@code Serializable}.
216     * It is equivalent to {@code tick(system(zone), Duration.ofSeconds(1))}.
217     *
218     * @param zone the time-zone to use to convert the instant to date-time, not null
219     * @return a clock that ticks in whole seconds using the specified zone, not null
220     */
221     public static Clock tickSeconds(ZoneId zone) {
222         return new TickClock(system(zone), NANOS_PER_SECOND);
223     }
224
225     /**
226     * Obtains a clock that returns the current instant ticking in whole minutes
227     * using best available system clock.
228     * <p>
229     * This clock will always have the nano-of-second and second-of-minute fields set to zero.
230     * This ensures that the visible time ticks in whole minutes.
231     * The underlying clock is the best available system clock, equivalent to
232     * using {@link #system(ZoneId)}.
233     * <p>
234     * Implementations may use a caching strategy for performance reasons.
235     * As such, it is possible that the start of the minute observed via this
236     * clock will be later than that observed directly via the underlying clock.
237     * <p>
238     * The returned implementation is immutable, thread-safe and {@code Serializable}.
239     * It is equivalent to {@code tick(system(zone), Duration.ofMinutes(1))}.
240     *
241     * @param zone the time-zone to use to convert the instant to date-time, not null
242     * @return a clock that ticks in whole minutes using the specified zone, not null

```



```
243     */
244     public static Clock tickMinutes(ZoneId zone) {
245         return new TickClock(system(zone), NANOS_PER_MINUTE);
246     }
247
248     /**
249     * Obtains a clock that returns instants from the specified clock truncated
250     * to the nearest occurrence of the specified duration.
251     * <p>
252     * This clock will only tick as per the specified duration. Thus, if the duration
253     * is half a second, the clock will return instants truncated to the half second.
254     * <p>
255     * The tick duration must be positive. If it has a part smaller than a whole
256     * millisecond, then the whole duration must divide into one second without
257     * leaving a remainder. All normal tick durations will match these criteria,
258     * including any multiple of hours, minutes, seconds and milliseconds, and
259     * sensible nanosecond durations, such as 20ns, 250,000ns and 500,000ns.
260     * <p>
261     * A duration of zero or one nanosecond would have no truncation effect.
262     * Passing one of these will return the underlying clock.
263     * <p>
264     * Implementations may use a caching strategy for performance reasons.
265     * As such, it is possible that the start of the requested duration observed
266     * via this clock will be later than that observed directly via the underlying clock.
267     * <p>
268     * The returned implementation is immutable, thread-safe and {@code Serializable}
269     * providing that the base clock is.
270     *
271     * @param baseClock the base clock to base the ticking clock on, not null
272     * @param tickDuration the duration of each visible tick, not negative, not null
273     * @return a clock that ticks in whole units of the duration, not null
274     * @throws IllegalArgumentException if the duration is negative, or has a
275     *     part smaller than a whole millisecond such that the whole duration is not
276     *     divisible into one second
277     * @throws ArithmeticException if the duration is too large to be represented as nanos
278     */
279     public static Clock tick(Clock baseClock, Duration tickDuration) {
280         Objects.requireNonNull(baseClock, "baseClock");
281         Objects.requireNonNull(tickDuration, "tickDuration");
282         if (tickDuration.isNegative()) {
283             throw new IllegalArgumentException("Tick duration must not be negative");
284         }
285         long tickNanos = tickDuration.toNanos();
286         if (tickNanos % 1000_000 == 0) {
287             // ok, no fraction of millisecond
288         } else if (1000_000_000 % tickNanos == 0) {
289             // ok, divides into one second without remainder
290         } else {
291             throw new IllegalArgumentException("Invalid tick duration");
292         }
293         if (tickNanos <= 1) {
294             return baseClock;
295         }
```

```

296         return new TickClock(baseClock, tickNanos);
297     }
298
299     //-----
300     /**
301      * Obtains a clock that always returns the same instant.
302      * <p>
303      * This clock simply returns the specified instant.
304      * As such, it is not a clock in the conventional sense.
305      * The main use case for this is in testing, where the fixed clock ensures
306      * tests are not dependent on the current clock.
307      * <p>
308      * The returned implementation is immutable, thread-safe and {@code Serializable}.
309      *
310      * @param fixedInstant the instant to use as the clock, not null
311      * @param zone the time-zone to use to convert the instant to date-time, not null
312      * @return a clock that always returns the same instant, not null
313      */
314     public static Clock fixed(Instant fixedInstant, ZoneId zone) {
315         Objects.requireNonNull(fixedInstant, "fixedInstant");
316         Objects.requireNonNull(zone, "zone");
317         return new FixedClock(fixedInstant, zone);
318     }
319
320     //-----
321     /**
322      * Obtains a clock that returns instants from the specified clock with the
323      * specified duration added
324      * <p>
325      * This clock wraps another clock, returning instants that are later by the
326      * specified duration. If the duration is negative, the instants will be
327      * earlier than the current date and time.
328      * The main use case for this is to simulate running in the future or in the past.
329      * <p>
330      * A duration of zero would have no offsetting effect.
331      * Passing zero will return the underlying clock.
332      * <p>
333      * The returned implementation is immutable, thread-safe and {@code Serializable}
334      * providing that the base clock is.
335      *
336      * @param baseClock the base clock to add the duration to, not null
337      * @param offsetDuration the duration to add, not null
338      * @return a clock based on the base clock with the duration added, not null
339      */
340     public static Clock offset(Clock baseClock, Duration offsetDuration) {
341         Objects.requireNonNull(baseClock, "baseClock");
342         Objects.requireNonNull(offsetDuration, "offsetDuration");
343         if (offsetDuration.equals(Duration.ZERO)) {
344             return baseClock;
345         }
346         return new OffsetClock(baseClock, offsetDuration);
347     }
348

```

```

349 //-----
350 /**
351  * Constructor accessible by subclasses.
352  */
353 protected Clock() {
354 }
355
356 //-----
357 /**
358  * Gets the time-zone being used to create dates and times.
359  * <p>
360  * A clock will typically obtain the current instant and then convert that
361  * to a date or time using a time-zone. This method returns the time-zone used.
362  *
363  * @return the time-zone being used to interpret instants, not null
364  */
365 public abstract ZoneId getZone();
366
367 /**
368  * Returns a copy of this clock with a different time-zone.
369  * <p>
370  * A clock will typically obtain the current instant and then convert that
371  * to a date or time using a time-zone. This method returns a clock with
372  * similar properties but using a different time-zone.
373  *
374  * @param zone the time-zone to change to, not null
375  * @return a clock based on this clock with the specified time-zone, not null
376  */
377 public abstract Clock withZone(ZoneId zone);
378
379 //-----
380 /**
381  * Gets the current millisecond instant of the clock.
382  * <p>
383  * This returns the millisecond-based instant, measured from 1970-01-01T00:00Z (UTC).
384  * This is equivalent to the definition of {@link System#currentTimeMillis()}.
385  * <p>
386  * Most applications should avoid this method and use {@link Instant} to represent
387  * an instant on the time-line rather than a raw millisecond value.
388  * This method is provided to allow the use of the clock in high performance use cases
389  * where the creation of an object would be unacceptable.
390  * <p>
391  * The default implementation currently calls {@link #instant}.
392  *
393  * @return the current millisecond instant from this clock, measured from
394  * the Java epoch of 1970-01-01T00:00Z (UTC), not null
395  * @throws DateTimeException if the instant cannot be obtained, not thrown by most
396  * implementations
397  */
398 public long millis() {
399     return instant().toEpochMilli();
400 }

```

```

401 //-----
402 /**
403  * Gets the current instant of the clock.
404  * <p>
405  * This returns an instant representing the current instant as defined by the clock.
406  *
407  * @return the current instant from this clock, not null
408  * @throws DateTimeException if the instant cannot be obtained, not thrown by most
         implementations
409  */
410 public abstract Instant instant();
411
412 //-----
413 /**
414  * Checks if this clock is equal to another clock.
415  * <p>
416  * Clocks should override this method to compare equals based on
417  * their state and to meet the contract of {@link Object#equals}.
418  * If not overridden, the behavior is defined by {@link Object#equals}
419  *
420  * @param obj the object to check, null returns false
421  * @return true if this is equal to the other clock
422  */
423 @Override
424 public boolean equals(Object obj) {
425     return super.equals(obj);
426 }
427
428 /**
429  * A hash code for this clock.
430  * <p>
431  * Clocks should override this method based on
432  * their state and to meet the contract of {@link Object#hashCode}.
433  * If not overridden, the behavior is defined by {@link Object#hashCode}
434  *
435  * @return a suitable hash code
436  */
437 @Override
438 public int hashCode() {
439     return super.hashCode();
440 }
441
442 //-----
443 /**
444  * Implementation of a clock that always returns the latest time from
445  * {@link System#currentTimeMillis()}.
446  */
447 static final class SystemClock extends Clock implements Serializable {
448     private static final long serialVersionUID = 6740630888130243051L;
449     private final ZoneId zone;
450
451     SystemClock(ZoneId zone) {
452         this.zone = zone;

```

```

453     }
454     @Override
455     public ZoneId getZone() {
456         return zone;
457     }
458     @Override
459     public Clock withZone(ZoneId zone) {
460         if (zone.equals(this.zone)) { // intentional NPE
461             return this;
462         }
463         return new SystemClock(zone);
464     }
465     @Override
466     public long millis() {
467         return System.currentTimeMillis();
468     }
469     @Override
470     public Instant instant() {
471         return Instant.ofEpochMilli(millis());
472     }
473     @Override
474     public boolean equals(Object obj) {
475         if (obj instanceof SystemClock) {
476             return zone.equals(((SystemClock) obj).zone);
477         }
478         return false;
479     }
480     @Override
481     public int hashCode() {
482         return zone.hashCode() + 1;
483     }
484     @Override
485     public String toString() {
486         return "SystemClock[" + zone + "]";
487     }
488 }
489
490 //-----
491 /**
492  * Implementation of a clock that always returns the same instant.
493  * This is typically used for testing.
494  */
495 static final class FixedClock extends Clock implements Serializable {
496     private static final long serialVersionUID = 7430389292664866958L;
497     private final Instant instant;
498     private final ZoneId zone;
499
500     FixedClock(Instant fixedInstant, ZoneId zone) {
501         this.instant = fixedInstant;
502         this.zone = zone;
503     }
504     @Override
505     public ZoneId getZone() {

```

```

506         return zone;
507     }
508     @Override
509     public Clock withZone(ZoneId zone) {
510         if (zone.equals(this.zone)) { // intentional NPE
511             return this;
512         }
513         return new FixedClock(instant, zone);
514     }
515     @Override
516     public long millis() {
517         return instant.toEpochMilli();
518     }
519     @Override
520     public Instant instant() {
521         return instant;
522     }
523     @Override
524     public boolean equals(Object obj) {
525         if (obj instanceof FixedClock) {
526             FixedClock other = (FixedClock) obj;
527             return instant.equals(other.instant) && zone.equals(other.zone);
528         }
529         return false;
530     }
531     @Override
532     public int hashCode() {
533         return instant.hashCode() ^ zone.hashCode();
534     }
535     @Override
536     public String toString() {
537         return "FixedClock[" + instant + "," + zone + "]";
538     }
539 }
540
541 //-----
542 /**
543  * Implementation of a clock that adds an offset to an underlying clock.
544  */
545 static final class OffsetClock extends Clock implements Serializable {
546     private static final long serialVersionUID = 2007484719125426256L;
547     private final Clock baseClock;
548     private final Duration offset;
549
550     OffsetClock(Clock baseClock, Duration offset) {
551         this.baseClock = baseClock;
552         this.offset = offset;
553     }
554     @Override
555     public ZoneId getZone() {
556         return baseClock.getZone();
557     }
558     @Override

```

```

559     public Clock withZone(ZoneId zone) {
560         if (zone.equals(baseClock.getZone())) { // intentional NPE
561             return this;
562         }
563         return new OffsetClock(baseClock.withZone(zone), offset);
564     }
565     @Override
566     public long millis() {
567         return Math.addExact(baseClock.millis(), offset.toMillis());
568     }
569     @Override
570     public Instant instant() {
571         return baseClock.instant().plus(offset);
572     }
573     @Override
574     public boolean equals(Object obj) {
575         if (obj instanceof OffsetClock) {
576             OffsetClock other = (OffsetClock) obj;
577             return baseClock.equals(other.baseClock) && offset.equals(other.offset);
578         }
579         return false;
580     }
581     @Override
582     public int hashCode() {
583         return baseClock.hashCode() ^ offset.hashCode();
584     }
585     @Override
586     public String toString() {
587         return "OffsetClock[" + baseClock + ", " + offset + "]";
588     }
589 }
590
591 //-----
592 /**
593  * Implementation of a clock that adds an offset to an underlying clock.
594  */
595 static final class TickClock extends Clock implements Serializable {
596     private static final long serialVersionUID = 6504659149906368850L;
597     private final Clock baseClock;
598     private final long tickNanos;
599
600     TickClock(Clock baseClock, long tickNanos) {
601         this.baseClock = baseClock;
602         this.tickNanos = tickNanos;
603     }
604     @Override
605     public ZoneId getZone() {
606         return baseClock.getZone();
607     }
608     @Override
609     public Clock withZone(ZoneId zone) {
610         if (zone.equals(baseClock.getZone())) { // intentional NPE
611             return this;

```

```

612     }
613     return new TickClock(baseClock.withZone(zone), tickNanos);
614 }
615 @Override
616 public long millis() {
617     long millis = baseClock.millis();
618     return millis - Math.floorMod(millis, tickNanos / 1000_000L);
619 }
620 @Override
621 public Instant instant() {
622     if ((tickNanos % 1000_000) == 0) {
623         long millis = baseClock.millis();
624         return Instant.ofEpochMilli(millis - Math.floorMod(millis, tickNanos / 1000_000L));
625     }
626     Instant instant = baseClock.instant();
627     long nanos = instant.getNano();
628     long adjust = Math.floorMod(nanos, tickNanos);
629     return instant.minusNanos(adjust);
630 }
631 @Override
632 public boolean equals(Object obj) {
633     if (obj instanceof TickClock) {
634         TickClock other = (TickClock) obj;
635         return baseClock.equals(other.baseClock) && tickNanos == other.tickNanos;
636     }
637     return false;
638 }
639 @Override
640 public int hashCode() {
641     return baseClock.hashCode() ^ ((int) (tickNanos ^ (tickNanos >>> 32)));
642 }
643 @Override
644 public String toString() {
645     return "TickClock[" + baseClock + ", " + Duration.ofNanos(tickNanos) + "]";
646 }
647 }
648
649 }

```

4.2 DateTimeException.java

Secuencia 51: java/time/DateTimeException.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *

```



```
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2008-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time;
63
64  /**
```

```

65  * Exception used to indicate a problem while calculating a date-time.
66  * <p>
67  * This exception is used to indicate problems with creating, querying
68  * and manipulating date-time objects.
69  *
70  * @implSpec
71  * This class is intended for use in a single thread.
72  *
73  * @since 1.8
74  */
75  public class DateTimeException extends RuntimeException {
76
77      /**
78       * Serialization version.
79       */
80      private static final long serialVersionUID = -1632418723876261839L;
81
82      /**
83       * Constructs a new date-time exception with the specified message.
84       *
85       * @param message the message to use for this exception, may be null
86       */
87      public DateTimeException(String message) {
88          super(message);
89      }
90
91      /**
92       * Constructs a new date-time exception with the specified message and cause.
93       *
94       * @param message the message to use for this exception, may be null
95       * @param cause the cause of the exception, may be null
96       */
97      public DateTimeException(String message, Throwable cause) {
98          super(message, cause);
99      }
100
101 }

```

4.3 DayOfWeek.java

Secuencia 52: java/time/DayOfWeek.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *

```

```
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time;
63
64  import static java.time.temporal.ChronoField.DAY_OF_WEEK;
65  import static java.time.temporal.ChronoUnit.DAYS;
```

```

66
67 import java.time.format.DateTimeFormatterBuilder;
68 import java.time.format.TextStyle;
69 import java.time.temporal.ChronoField;
70 import java.time.temporal.Temporal;
71 import java.time.temporal.TemporalAccessor;
72 import java.time.temporal.TemporalAdjuster;
73 import java.time.temporal.TemporalField;
74 import java.time.temporal.TemporalQueries;
75 import java.time.temporal.TemporalQuery;
76 import java.time.temporal.UnsupportedTemporalTypeException;
77 import java.time.temporal.ValueRange;
78 import java.time.temporal.WeekFields;
79 import java.util.Locale;
80
81 /**
82  * A day-of-week, such as 'Tuesday'.
83  * <p>
84  * {@code DayOfWeek} is an enum representing the 7 days of the week -
85  * Monday, Tuesday, Wednesday, Thursday, Friday, Saturday and Sunday.
86  * <p>
87  * In addition to the textual enum name, each day-of-week has an {@code int} value.
88  * The {@code int} value follows the ISO-8601 standard, from 1 (Monday) to 7 (Sunday).
89  * It is recommended that applications use the enum rather than the {@code int} value
90  * to ensure code clarity.
91  * <p>
92  * This enum provides access to the localized textual form of the day-of-week.
93  * Some locales also assign different numeric values to the days, declaring
94  * Sunday to have the value 1, however this class provides no support for this.
95  * See {@link WeekFields} for localized week-numbering.
96  * <p>
97  * <b>Do not use {@code ordinal()} to obtain the numeric representation of {@code DayOfWeek}.
98  * Use {@code getValue()} instead.</b>
99  * <p>
100  * This enum represents a common concept that is found in many calendar systems.
101  * As such, this enum may be used by any calendar system that has the day-of-week
102  * concept defined exactly equivalent to the ISO calendar system.
103  *
104  * @implSpec
105  * This is an immutable and thread-safe enum.
106  *
107  * @since 1.8
108  */
109 public enum DayOfWeek implements TemporalAccessor, TemporalAdjuster {
110
111     /**
112      * The singleton instance for the day-of-week of Monday.
113      * This has the numeric value of {@code 1}.
114      */
115     MONDAY,
116
117     /**
118      * The singleton instance for the day-of-week of Tuesday.
119      * This has the numeric value of {@code 2}.

```

```

119     */
120     TUESDAY,
121     /**
122      * The singleton instance for the day-of-week of Wednesday.
123      * This has the numeric value of {@code 3}.
124      */
125     WEDNESDAY,
126     /**
127      * The singleton instance for the day-of-week of Thursday.
128      * This has the numeric value of {@code 4}.
129      */
130     THURSDAY,
131     /**
132      * The singleton instance for the day-of-week of Friday.
133      * This has the numeric value of {@code 5}.
134      */
135     FRIDAY,
136     /**
137      * The singleton instance for the day-of-week of Saturday.
138      * This has the numeric value of {@code 6}.
139      */
140     SATURDAY,
141     /**
142      * The singleton instance for the day-of-week of Sunday.
143      * This has the numeric value of {@code 7}.
144      */
145     SUNDAY;
146     /**
147      * Private cache of all the constants.
148      */
149     private static final DayOfWeek[] ENUMS = DayOfWeek.values();
150
151     //-----
152     /**
153      * Obtains an instance of {@code DayOfWeek} from an {@code int} value.
154      * <p>
155      * {@code DayOfWeek} is an enum representing the 7 days of the week.
156      * This factory allows the enum to be obtained from the {@code int} value.
157      * The {@code int} value follows the ISO-8601 standard, from 1 (Monday) to 7 (Sunday).
158      *
159      * @param dayOfWeek the day-of-week to represent, from 1 (Monday) to 7 (Sunday)
160      * @return the day-of-week singleton, not null
161      * @throws DateTimeException if the day-of-week is invalid
162      */
163     public static DayOfWeek of(int dayOfWeek) {
164         if (dayOfWeek < 1 || dayOfWeek > 7) {
165             throw new DateTimeException("Invalid value for DayOfWeek: " + dayOfWeek);
166         }
167         return ENUMS[dayOfWeek - 1];
168     }
169
170     //-----
171     /**

```

```

172 * Obtains an instance of {@code DayOfWeek} from a temporal object.
173 * <p>
174 * This obtains a day-of-week based on the specified temporal.
175 * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
176 * which this factory converts to an instance of {@code DayOfWeek}.
177 * <p>
178 * The conversion extracts the {@link ChronoField#DAY_OF_WEEK DAY_OF_WEEK} field.
179 * <p>
180 * This method matches the signature of the functional interface {@link TemporalQuery}
181 * allowing it to be used as a query via method reference, {@code DayOfWeek::from}.
182 *
183 * @param temporal the temporal object to convert, not null
184 * @return the day-of-week, not null
185 * @throws DateTimeException if unable to convert to a {@code DayOfWeek}
186 */
187 public static DayOfWeek from(TemporalAccessor temporal) {
188     if (temporal instanceof DayOfWeek) {
189         return (DayOfWeek) temporal;
190     }
191     try {
192         return of(temporal.get(DAY_OF_WEEK));
193     } catch (DateTimeException ex) {
194         throw new DateTimeException("Unable to obtain DayOfWeek from TemporalAccessor: " +
195             temporal + " of type " + temporal.getClass().getName(), ex);
196     }
197 }
198
199 //-----
200 /**
201 * Gets the day-of-week {@code int} value.
202 * <p>
203 * The values are numbered following the ISO-8601 standard, from 1 (Monday) to 7 (Sunday).
204 * See {@link java.time.temporal.WeekFields#dayOfWeek()} for localized week-numbering.
205 *
206 * @return the day-of-week, from 1 (Monday) to 7 (Sunday)
207 */
208 public int getValue() {
209     return ordinal() + 1;
210 }
211
212 //-----
213 /**
214 * Gets the textual representation, such as 'Mon' or 'Friday'.
215 * <p>
216 * This returns the textual name used to identify the day-of-week,
217 * suitable for presentation to the user.
218 * The parameters control the style of the returned text and the locale.
219 * <p>
220 * If no textual mapping is found then the {@link #getValue() numeric value} is returned.
221 *
222 * @param style the length of the text required, not null
223 * @param locale the locale to use, not null
224 * @return the text value of the day-of-week, not null

```

```

225     */
226     public String getDisplayName(TextStyle style, Locale locale) {
227         return new DateTimeFormatterBuilder().appendText(DAY_OF_WEEK, style).toFormatter(locale).
            format(this);
228     }
229
230     //-----
231     /**
232      * Checks if the specified field is supported.
233      * <p>
234      * This checks if this day-of-week can be queried for the specified field.
235      * If false, then calling the {@link #range(TemporalField) range} and
236      * {@link #get(TemporalField) get} methods will throw an exception.
237      * <p>
238      * If the field is {@link ChronoField#DAY_OF_WEEK DAY_OF_WEEK} then
239      * this method returns true.
240      * All other {@code ChronoField} instances will return false.
241      * <p>
242      * If the field is not a {@code ChronoField}, then the result of this method
243      * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
244      * passing {@code this} as the argument.
245      * Whether the field is supported is determined by the field.
246      *
247      * @param field the field to check, null returns false
248      * @return true if the field is supported on this day-of-week, false if not
249      */
250     @Override
251     public boolean isSupported(TemporalField field) {
252         if (field instanceof ChronoField) {
253             return field == DAY_OF_WEEK;
254         }
255         return field != null && field.isSupportedBy(this);
256     }
257
258     /**
259      * Gets the range of valid values for the specified field.
260      * <p>
261      * The range object expresses the minimum and maximum valid values for a field.
262      * This day-of-week is used to enhance the accuracy of the returned range.
263      * If it is not possible to return the range, because the field is not supported
264      * or for some other reason, an exception is thrown.
265      * <p>
266      * If the field is {@link ChronoField#DAY_OF_WEEK DAY_OF_WEEK} then the
267      * range of the day-of-week, from 1 to 7, will be returned.
268      * All other {@code ChronoField} instances will throw an {@code
269      * UnsupportedTemporalTypeException}.
270      * <p>
271      * If the field is not a {@code ChronoField}, then the result of this method
272      * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
273      * passing {@code this} as the argument.
274      * Whether the range can be obtained is determined by the field.
275      *
276      * @param field the field to query the range for, not null

```

```

276     * @return the range of valid values for the field, not null
277     * @throws DateTimeException if the range for the field cannot be obtained
278     * @throws UnsupportedTemporalTypeException if the field is not supported
279     */
280     @Override
281     public ValueRange range(TemporalField field) {
282         if (field == DAY_OF_WEEK) {
283             return field.range();
284         }
285         return TemporalAccessor.super.range(field);
286     }
287
288     /**
289     * Gets the value of the specified field from this day-of-week as an {@code int}.
290     * <p>
291     * This queries this day-of-week for the value of the specified field.
292     * The returned value will always be within the valid range of values for the field.
293     * If it is not possible to return the value, because the field is not supported
294     * or for some other reason, an exception is thrown.
295     * <p>
296     * If the field is {@link ChronoField#DAY_OF_WEEK DAY_OF_WEEK} then the
297     * value of the day-of-week, from 1 to 7, will be returned.
298     * All other {@code ChronoField} instances will throw an {@code
299         UnsupportedTemporalTypeException}.
300     * <p>
301     * If the field is not a {@code ChronoField}, then the result of this method
302     * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
303     * passing {@code this} as the argument. Whether the value can be obtained,
304     * and what the value represents, is determined by the field.
305     *
306     * @param field the field to get, not null
307     * @return the value for the field, within the valid range of values
308     * @throws DateTimeException if a value for the field cannot be obtained or
309     *         the value is outside the range of valid values for the field
310     * @throws UnsupportedTemporalTypeException if the field is not supported or
311     *         the range of values exceeds an {@code int}
312     * @throws ArithmeticException if numeric overflow occurs
313     */
314     @Override
315     public int get(TemporalField field) {
316         if (field == DAY_OF_WEEK) {
317             return getValue();
318         }
319         return TemporalAccessor.super.get(field);
320     }
321
322     /**
323     * Gets the value of the specified field from this day-of-week as a {@code long}.
324     * <p>
325     * This queries this day-of-week for the value of the specified field.
326     * If it is not possible to return the value, because the field is not supported
327     * or for some other reason, an exception is thrown.
328     * <p>

```



```

328 * If the field is {@link ChronoField#DAY_OF_WEEK DAY_OF_WEEK} then the
329 * value of the day-of-week, from 1 to 7, will be returned.
330 * All other {@code ChronoField} instances will throw an {@code
    UnsupportedTemporalTypeException}.
331 * <p>
332 * If the field is not a {@code ChronoField}, then the result of this method
333 * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
334 * passing {@code this} as the argument. Whether the value can be obtained,
335 * and what the value represents, is determined by the field.
336 *
337 * @param field the field to get, not null
338 * @return the value for the field
339 * @throws DateTimeException if a value for the field cannot be obtained
340 * @throws UnsupportedTemporalTypeException if the field is not supported
341 * @throws ArithmeticException if numeric overflow occurs
342 */
343 @Override
344 public long getLong(TemporalField field) {
345     if (field == DAY_OF_WEEK) {
346         return getValue();
347     } else if (field instanceof ChronoField) {
348         throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
349     }
350     return field.getFrom(this);
351 }
352
353 //-----
354 /**
355  * Returns the day-of-week that is the specified number of days after this one.
356  * <p>
357  * The calculation rolls around the end of the week from Sunday to Monday.
358  * The specified period may be negative.
359  * <p>
360  * This instance is immutable and unaffected by this method call.
361  *
362  * @param days the days to add, positive or negative
363  * @return the resulting day-of-week, not null
364  */
365 public DayOfWeek plus(long days) {
366     int amount = (int) (days % 7);
367     return ENUMS[(ordinal() + (amount + 7)) % 7];
368 }
369
370 /**
371  * Returns the day-of-week that is the specified number of days before this one.
372  * <p>
373  * The calculation rolls around the start of the year from Monday to Sunday.
374  * The specified period may be negative.
375  * <p>
376  * This instance is immutable and unaffected by this method call.
377  *
378  * @param days the days to subtract, positive or negative
379  * @return the resulting day-of-week, not null

```

```

380     */
381     public DayOfWeek minus(long days) {
382         return plus(-(days % 7));
383     }
384
385     //-----
386     /**
387      * Queries this day-of-week using the specified query.
388      * <p>
389      * This queries this day-of-week using the specified query strategy object.
390      * The {@code TemporalQuery} object defines the logic to be used to
391      * obtain the result. Read the documentation of the query to understand
392      * what the result of this method will be.
393      * <p>
394      * The result of this method is obtained by invoking the
395      * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
396      * specified query passing {@code this} as the argument.
397      *
398      * @param <R> the type of the result
399      * @param query the query to invoke, not null
400      * @return the query result, null may be returned (defined by the query)
401      * @throws DateTimeException if unable to query (defined by the query)
402      * @throws ArithmeticException if numeric overflow occurs (defined by the query)
403      */
404     @SuppressWarnings("unchecked")
405     @Override
406     public <R> R query(TemporalQuery<R> query) {
407         if (query == TemporalQueries.precision()) {
408             return (R) DAYS;
409         }
410         return TemporalAccessor.super.query(query);
411     }
412
413     /**
414      * Adjusts the specified temporal object to have this day-of-week.
415      * <p>
416      * This returns a temporal object of the same observable type as the input
417      * with the day-of-week changed to be the same as this.
418      * <p>
419      * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
420      * passing {@link ChronoField#DAY_OF_WEEK} as the field.
421      * Note that this adjusts forwards or backwards within a Monday to Sunday week.
422      * See {@link java.time.temporal.WeekFields#dayOfWeek()} for localized week start days.
423      * See {@code TemporalAdjuster} for other adjusters with more control,
424      * such as {@code next(MONDAY)}.
425      * <p>
426      * In most cases, it is clearer to reverse the calling pattern by using
427      * {@link Temporal#with(TemporalAdjuster)}:
428      * <pre>
429      * // these two lines are equivalent, but the second approach is recommended
430      * temporal = thisDayOfWeek.adjustInto(temporal);
431      * temporal = temporal.with(thisDayOfWeek);
432      * </pre>

```

```

433 * <p>
434 * For example, given a date that is a Wednesday, the following are output:
435 * <pre>
436 *   dateOnWed.with(MONDAY);    // two days earlier
437 *   dateOnWed.with(TUESDAY);   // one day earlier
438 *   dateOnWed.with(WEDNESDAY); // same date
439 *   dateOnWed.with(THURSDAY);  // one day later
440 *   dateOnWed.with(FRIDAY);    // two days later
441 *   dateOnWed.with(SATURDAY);  // three days later
442 *   dateOnWed.with(SUNDAY);    // four days later
443 * </pre>
444 * <p>
445 * This instance is immutable and unaffected by this method call.
446 *
447 * @param temporal the target object to be adjusted, not null
448 * @return the adjusted object, not null
449 * @throws DateTimeException if unable to make the adjustment
450 * @throws ArithmeticException if numeric overflow occurs
451 */
452 @Override
453 public Temporal adjustInto(Temporal temporal) {
454     return temporal.with(DAY_OF_WEEK, getValue());
455 }
456
457 }

```

4.4 Duration.java

Secuencia 53: java/time/Duration.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */

```

```
25
26 /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62 package java.time;
63
64 import static java.time.LocalTime.NANOS_PER_SECOND;
65 import static java.time.LocalTime.SECONDS_PER_DAY;
66 import static java.time.LocalTime.SECONDS_PER_HOUR;
67 import static java.time.LocalTime.SECONDS_PER_MINUTE;
68 import static java.time.temporal.ChronoField.NANO_OF_SECOND;
69 import static java.time.temporal.ChronoUnit.DAYS;
70 import static java.time.temporal.ChronoUnit.NANOS;
71 import static java.time.temporal.ChronoUnit.SECONDS;
72
73 import java.io.DataInput;
74 import java.io.DataOutput;
75 import java.io.IOException;
76 import java.io.InvalidObjectException;
77 import java.io.ObjectInputStream;
```

```

78 import java.io.Serializable;
79 import java.math.BigDecimal;
80 import java.math.BigInteger;
81 import java.math.RoundingMode;
82 import java.time.format.DateTimeParseException;
83 import java.time.temporal.ChronoField;
84 import java.time.temporal.ChronoUnit;
85 import java.time.temporal.Temporal;
86 import java.time.temporal.TemporalAmount;
87 import java.time.temporal.TemporalUnit;
88 import java.time.temporal.UnsupportedTemporalTypeException;
89 import java.util.Arrays;
90 import java.util.Collections;
91 import java.util.List;
92 import java.util.Objects;
93 import java.util.regex.Matcher;
94 import java.util.regex.Pattern;
95
96 /**
97  * A time-based amount of time, such as '34.5 seconds'.
98  * <p>
99  * This class models a quantity or amount of time in terms of seconds and nanoseconds.
100  * It can be accessed using other duration-based units, such as minutes and hours.
101  * In addition, the {@link ChronoUnit#DAYS DAYS} unit can be used and is treated as
102  * exactly equal to 24 hours, thus ignoring daylight savings effects.
103  * See {@link Period} for the date-based equivalent to this class.
104  * <p>
105  * A physical duration could be of infinite length.
106  * For practicality, the duration is stored with constraints similar to {@link Instant}.
107  * The duration uses nanosecond resolution with a maximum value of the seconds that can
108  * be held in a {@code long}. This is greater than the current estimated age of the universe.
109  * <p>
110  * The range of a duration requires the storage of a number larger than a {@code long}.
111  * To achieve this, the class stores a {@code long} representing seconds and an {@code int}
112  * representing nanosecond-of-second, which will always be between 0 and 999,999,999.
113  * The model is of a directed duration, meaning that the duration may be negative.
114  * <p>
115  * The duration is measured in "seconds", but these are not necessarily identical to
116  * the scientific "SI second" definition based on atomic clocks.
117  * This difference only impacts durations measured near a leap-second and should not affect
118  * most applications.
119  * See {@link Instant} for a discussion as to the meaning of the second and time-scales.
120  *
121  * <p>
122  * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
123  * class; use of identity-sensitive operations (including reference equality
124  * ({@code ==}), identity hash code, or synchronization) on instances of
125  * {@code Duration} may have unpredictable results and should be avoided.
126  * The {@code equals} method should be used for comparisons.
127  *
128  * @implSpec
129  * This class is immutable and thread-safe.
130  *

```

```

131  * @since 1.8
132  */
133  public final class Duration
134      implements TemporalAmount, Comparable<Duration>, Serializable {
135
136      /**
137       * Constant for a duration of zero.
138       */
139      public static final Duration ZERO = new Duration(0, 0);
140      /**
141       * Serialization version.
142       */
143      private static final long serialVersionUID = 3078945930695997490L;
144      /**
145       * Constant for nanos per second.
146       */
147      private static final BigInteger BI_NANOS_PER_SECOND = BigInteger.valueOf(NANOS_PER_SECOND);
148      /**
149       * The pattern for parsing.
150       */
151      private static final Pattern PATTERN =
152          Pattern.compile("([-+]?)P(?:([-+]?[0-9]+)D)?" +
153              "(T(?:([-+]?[0-9]+)H)?(?:([-+]?[0-9]+)M)?(?:([-+]?[0-9]+)(?:[.](?:[0-9]{0,9}))?)?)?)?",
154              Pattern.CASE_INSENSITIVE);
155
156      /**
157       * The number of seconds in the duration.
158       */
159      private final long seconds;
160      /**
161       * The number of nanoseconds in the duration, expressed as a fraction of the
162       * number of seconds. This is always positive, and never exceeds 999,999,999.
163       */
164      private final int nanos;
165
166      //-----
167      /**
168       * Obtains a {@code Duration} representing a number of standard 24 hour days.
169       * <p>
170       * The seconds are calculated based on the standard definition of a day,
171       * where each day is 86400 seconds which implies a 24 hour day.
172       * The nanosecond in second field is set to zero.
173       *
174       * @param days the number of days, positive or negative
175       * @return a {@code Duration}, not null
176       * @throws ArithmeticException if the input days exceeds the capacity of {@code Duration}
177       */
178      public static Duration ofDays(long days) {
179          return create(Math.multiplyExact(days, SECONDS_PER_DAY), 0);
180      }
181
182      /**

```

```

183     * Obtains a {@code Duration} representing a number of standard hours.
184     * <p>
185     * The seconds are calculated based on the standard definition of an hour,
186     * where each hour is 3600 seconds.
187     * The nanosecond in second field is set to zero.
188     *
189     * @param hours the number of hours, positive or negative
190     * @return a {@code Duration}, not null
191     * @throws ArithmeticException if the input hours exceeds the capacity of {@code Duration}
192     */
193     public static Duration ofHours(long hours) {
194         return create(Math.multiplyExact(hours, SECONDS_PER_HOUR), 0);
195     }
196
197     /**
198     * Obtains a {@code Duration} representing a number of standard minutes.
199     * <p>
200     * The seconds are calculated based on the standard definition of a minute,
201     * where each minute is 60 seconds.
202     * The nanosecond in second field is set to zero.
203     *
204     * @param minutes the number of minutes, positive or negative
205     * @return a {@code Duration}, not null
206     * @throws ArithmeticException if the input minutes exceeds the capacity of {@code Duration}
207     */
208     public static Duration ofMinutes(long minutes) {
209         return create(Math.multiplyExact(minutes, SECONDS_PER_MINUTE), 0);
210     }
211
212     //-----
213     /**
214     * Obtains a {@code Duration} representing a number of seconds.
215     * <p>
216     * The nanosecond in second field is set to zero.
217     *
218     * @param seconds the number of seconds, positive or negative
219     * @return a {@code Duration}, not null
220     */
221     public static Duration ofSeconds(long seconds) {
222         return create(seconds, 0);
223     }
224
225     /**
226     * Obtains a {@code Duration} representing a number of seconds and an
227     * adjustment in nanoseconds.
228     * <p>
229     * This method allows an arbitrary number of nanoseconds to be passed in.
230     * The factory will alter the values of the second and nanosecond in order
231     * to ensure that the stored nanosecond is in the range 0 to 999,999,999.
232     * For example, the following will result in the exactly the same duration:
233     * <pre>
234     * Duration.ofSeconds(3, 1);
235     * Duration.ofSeconds(4, -999_999_999);

```

```

236 * Duration.ofSeconds(2, 1000_000_001);
237 * </pre>
238 *
239 * @param seconds the number of seconds, positive or negative
240 * @param nanoAdjustment the nanosecond adjustment to the number of seconds, positive or
    negative
241 * @return a {@code Duration}, not null
242 * @throws ArithmeticException if the adjustment causes the seconds to exceed the capacity of {
    @code Duration}
243 */
244 public static Duration ofSeconds(long seconds, long nanoAdjustment) {
245     long secs = Math.addExact(seconds, Math.floorDiv(nanoAdjustment, NANOS_PER_SECOND));
246     int nos = (int) Math.floorMod(nanoAdjustment, NANOS_PER_SECOND);
247     return create(secs, nos);
248 }
249
250 //-----
251 /**
252  * Obtains a {@code Duration} representing a number of milliseconds.
253  * <p>
254  * The seconds and nanoseconds are extracted from the specified milliseconds.
255  *
256  * @param millis the number of milliseconds, positive or negative
257  * @return a {@code Duration}, not null
258  */
259 public static Duration ofMillis(long millis) {
260     long secs = millis / 1000;
261     int mos = (int) (millis % 1000);
262     if (mos < 0) {
263         mos += 1000;
264         secs--;
265     }
266     return create(secs, mos * 1000_000);
267 }
268
269 //-----
270 /**
271  * Obtains a {@code Duration} representing a number of nanoseconds.
272  * <p>
273  * The seconds and nanoseconds are extracted from the specified nanoseconds.
274  *
275  * @param nanos the number of nanoseconds, positive or negative
276  * @return a {@code Duration}, not null
277  */
278 public static Duration ofNanos(long nanos) {
279     long secs = nanos / NANOS_PER_SECOND;
280     int nos = (int) (nanos % NANOS_PER_SECOND);
281     if (nos < 0) {
282         nos += NANOS_PER_SECOND;
283         secs--;
284     }
285     return create(secs, nos);
286 }

```



```

287
288 //-----
289 /**
290  * Obtains a {@code Duration} representing an amount in the specified unit.
291  * <p>
292  * The parameters represent the two parts of a phrase like '6 Hours'. For example:
293  * <pre>
294  * Duration.of(3, SECONDS);
295  * Duration.of(465, HOURS);
296  * </pre>
297  * Only a subset of units are accepted by this method.
298  * The unit must either have an {@link TemporalUnit#isDurationEstimated() exact duration}
299  * or
300  * be {@link ChronoUnit#DAYS} which is treated as 24 hours. Other units throw an exception.
301  *
302  * @param amount the amount of the duration, measured in terms of the unit, positive or
303  *               negative
304  * @param unit the unit that the duration is measured in, must have an exact duration, not
305  *             null
306  * @return a {@code Duration}, not null
307  * @throws DateTimeException if the period unit has an estimated duration
308  * @throws ArithmeticException if a numeric overflow occurs
309  */
310
311 public static Duration of(long amount, TemporalUnit unit) {
312     return ZERO.plus(amount, unit);
313 }
314
315 //-----
316 /**
317  * Obtains an instance of {@code Duration} from a temporal amount.
318  * <p>
319  * This obtains a duration based on the specified amount.
320  * A {@code TemporalAmount} represents an amount of time, which may be
321  * date-based or time-based, which this factory extracts to a duration.
322  * <p>
323  * The conversion loops around the set of units from the amount and uses
324  * the {@link TemporalUnit#getDuration() duration} of the unit to
325  * calculate the total {@code Duration}.
326  * Only a subset of units are accepted by this method. The unit must either
327  * have an {@link TemporalUnit#isDurationEstimated() exact duration}
328  * or be {@link ChronoUnit#DAYS} which is treated as 24 hours.
329  * If any other units are found then an exception is thrown.
330  *
331  * @param amount the temporal amount to convert, not null
332  * @return the equivalent duration, not null
333  * @throws DateTimeException if unable to convert to a {@code Duration}
334  * @throws ArithmeticException if numeric overflow occurs
335  */
336 public static Duration from(TemporalAmount amount) {
337     Objects.requireNonNull(amount, "amount");
338     Duration duration = ZERO;
339     for (TemporalUnit unit : amount.getUnits()) {
340         duration = duration.plus(amount.get(unit), unit);
341     }
342 }

```

```

337     }
338     return duration;
339 }
340
341 //-----
342 /**
343  * Obtains a {@code Duration} from a text string such as {@code PnDTnHnMn.nS}.
344  * <p>
345  * This will parse a textual representation of a duration, including the
346  * string produced by {@code toString()}. The formats accepted are based
347  * on the ISO-8601 duration format {@code PnDTnHnMn.nS} with days
348  * considered to be exactly 24 hours.
349  * <p>
350  * The string starts with an optional sign, denoted by the ASCII negative
351  * or positive symbol. If negative, the whole period is negated.
352  * The ASCII letter "P" is next in upper or lower case.
353  * There are then four sections, each consisting of a number and a suffix.
354  * The sections have suffixes in ASCII of "D", "H", "M" and "S" for
355  * days, hours, minutes and seconds, accepted in upper or lower case.
356  * The suffixes must occur in order. The ASCII letter "T" must occur before
357  * the first occurrence, if any, of an hour, minute or second section.
358  * At least one of the four sections must be present, and if "T" is present
359  * there must be at least one section after the "T".
360  * The number part of each section must consist of one or more ASCII digits.
361  * The number may be prefixed by the ASCII negative or positive symbol.
362  * The number of days, hours and minutes must parse to an {@code long}.
363  * The number of seconds must parse to an {@code long} with optional fraction.
364  * The decimal point may be either a dot or a comma.
365  * The fractional part may have from zero to 9 digits.
366  * <p>
367  * The leading plus/minus sign, and negative values for other units are
368  * not part of the ISO-8601 standard.
369  * <p>
370  * Examples:
371  * <pre>
372  *     "PT20.345S" -- parses as "20.345 seconds"
373  *     "PT15M"     -- parses as "15 minutes" (where a minute is 60 seconds)
374  *     "PT10H"     -- parses as "10 hours" (where an hour is 3600 seconds)
375  *     "P2D"       -- parses as "2 days" (where a day is 24 hours or 86400 seconds)
376  *     "P2DT3H4M"  -- parses as "2 days, 3 hours and 4 minutes"
377  *     "P-6H3M"    -- parses as "-6 hours and +3 minutes"
378  *     "-P6H3M"    -- parses as "-6 hours and -3 minutes"
379  *     "-P-6H+3M"  -- parses as "+6 hours and -3 minutes"
380  * </pre>
381  *
382  * @param text the text to parse, not null
383  * @return the parsed duration, not null
384  * @throws DateTimeParseException if the text cannot be parsed to a duration
385  */
386 public static Duration parse(CharSequence text) {
387     Objects.requireNonNull(text, "text");
388     Matcher matcher = PATTERN.matcher(text);
389     if (matcher.matches()) {

```

```

390         // check for letter T but no time sections
391         if ("T".equals(matcher.group(3)) == false) {
392             boolean negate = "-".equals(matcher.group(1));
393             String dayMatch = matcher.group(2);
394             String hourMatch = matcher.group(4);
395             String minuteMatch = matcher.group(5);
396             String secondMatch = matcher.group(6);
397             String fractionMatch = matcher.group(7);
398             if (dayMatch != null || hourMatch != null || minuteMatch != null || secondMatch !=
399                 null) {
400                 long daysAsSecs = parseNumber(text, dayMatch, SECONDS_PER_DAY, "days");
401                 long hoursAsSecs = parseNumber(text, hourMatch, SECONDS_PER_HOUR, "hours");
402                 long minsAsSecs = parseNumber(text, minuteMatch, SECONDS_PER_MINUTE, "minutes");
403                 ;
404                 long seconds = parseNumber(text, secondMatch, 1, "seconds");
405                 int nanos = parseFraction(text, fractionMatch, seconds < 0 ? -1 : 1);
406                 try {
407                     return create(negate, daysAsSecs, hoursAsSecs, minsAsSecs, seconds, nanos);
408                 } catch (ArithmeticException ex) {
409                     throw (DateTimeParseException) new DateTimeParseException("Text cannot be
410                         parsed to a Duration: overflow", text, 0).initCause(ex);
411                 }
412             }
413         }
414         throw new DateTimeParseException("Text cannot be parsed to a Duration", text, 0);
415     }
416
417     private static long parseNumber(CharSequence text, String parsed, int multiplier, String
418         errorText) {
419         // regex limits to [-+]?[0-9]+
420         if (parsed == null) {
421             return 0;
422         }
423         try {
424             long val = Long.parseLong(parsed);
425             return Math.multiplyExact(val, multiplier);
426         } catch (NumberFormatException | ArithmeticException ex) {
427             throw (DateTimeParseException) new DateTimeParseException("Text cannot be parsed to a
428                 Duration: " + errorText, text, 0).initCause(ex);
429         }
430     }
431
432     private static int parseFraction(CharSequence text, String parsed, int negate) {
433         // regex limits to [0-9]{0,9}
434         if (parsed == null || parsed.length() == 0) {
435             return 0;
436         }
437         try {
438             parsed = (parsed + "000000000").substring(0, 9);
439             return Integer.parseInt(parsed) * negate;
440         } catch (NumberFormatException | ArithmeticException ex) {
441             throw (DateTimeParseException) new DateTimeParseException("Text cannot be parsed to a

```

```

        Duration: fraction", text, 0).initCause(ex);
438     }
439 }
440
441 private static Duration create(boolean negate, long daysAsSecs, long hoursAsSecs, long
    minsAsSecs, long secs, int nanos) {
442     long seconds = Math.addExact(daysAsSecs, Math.addExact(hoursAsSecs, Math.addExact(
        minsAsSecs, secs)));
443     if (negate) {
444         return ofSeconds(seconds, nanos).negated();
445     }
446     return ofSeconds(seconds, nanos);
447 }
448
449 //-----
450 /**
451  * Obtains a {@code Duration} representing the duration between two temporal objects.
452  * <p>
453  * This calculates the duration between two temporal objects. If the objects
454  * are of different types, then the duration is calculated based on the type
455  * of the first object. For example, if the first argument is a {@code LocalDateTime}
456  * then the second argument is converted to a {@code LocalDateTime}.
457  * <p>
458  * The specified temporal objects must support the {@link ChronoUnit#SECONDS SECONDS} unit.
459  * For full accuracy, either the {@link ChronoUnit#NANOS NANOS} unit or the
460  * {@link ChronoField#NANO_OF_SECOND NANO_OF_SECOND} field should be supported.
461  * <p>
462  * The result of this method can be a negative period if the end is before the start.
463  * To guarantee to obtain a positive duration call {@link #abs()} on the result.
464  *
465  * @param startInclusive the start instant, inclusive, not null
466  * @param endExclusive the end instant, exclusive, not null
467  * @return a {@code Duration}, not null
468  * @throws DateTimeException if the seconds between the temporals cannot be obtained
469  * @throws ArithmeticException if the calculation exceeds the capacity of {@code Duration}
470  */
471 public static Duration between(Temporal startInclusive, Temporal endExclusive) {
472     try {
473         return ofNanos(startInclusive.until(endExclusive, NANOS));
474     } catch (DateTimeException | ArithmeticException ex) {
475         long secs = startInclusive.until(endExclusive, SECONDS);
476         long nanos;
477         try {
478             nanos = endExclusive.getLong(NANO_OF_SECOND) - startInclusive.getLong(
                NANO_OF_SECOND);
479             if (secs > 0 && nanos < 0) {
480                 secs++;
481             } else if (secs < 0 && nanos > 0) {
482                 secs--;
483             }
484         } catch (DateTimeException ex2) {
485             nanos = 0;
486         }

```

```

487         return ofSeconds(secs, nanos);
488     }
489 }
490
491 //-----
492 /**
493  * Obtains an instance of {@code Duration} using seconds and nanoseconds.
494  *
495  * @param seconds the length of the duration in seconds, positive or negative
496  * @param nanoAdjustment the nanosecond adjustment within the second, from 0 to 999,999,999
497  */
498 private static Duration create(long seconds, int nanoAdjustment) {
499     if ((seconds | nanoAdjustment) == 0) {
500         return ZERO;
501     }
502     return new Duration(seconds, nanoAdjustment);
503 }
504
505 /**
506  * Constructs an instance of {@code Duration} using seconds and nanoseconds.
507  *
508  * @param seconds the length of the duration in seconds, positive or negative
509  * @param nanos the nanoseconds within the second, from 0 to 999,999,999
510  */
511 private Duration(long seconds, int nanos) {
512     super();
513     this.seconds = seconds;
514     this.nanos = nanos;
515 }
516
517 //-----
518 /**
519  * Gets the value of the requested unit.
520  * <p>
521  * This returns a value for each of the two supported units,
522  * {@link ChronoUnit#SECONDS SECONDS} and {@link ChronoUnit#NANOS NANOS}.
523  * All other units throw an exception.
524  *
525  * @param unit the {@code TemporalUnit} for which to return the value
526  * @return the long value of the unit
527  * @throws DateTimeException if the unit is not supported
528  * @throws UnsupportedTemporalTypeException if the unit is not supported
529  */
530 @Override
531 public long get(TemporalUnit unit) {
532     if (unit == SECONDS) {
533         return seconds;
534     } else if (unit == NANOS) {
535         return nanos;
536     } else {
537         throw new UnsupportedTemporalTypeException("Unsupported unit: " + unit);
538     }
539 }

```

```

540
541 /**
542  * Gets the set of units supported by this duration.
543  * <p>
544  * The supported units are {@link ChronoUnit#SECONDS SECONDS},
545  * and {@link ChronoUnit#NANOS NANOS}.
546  * They are returned in the order seconds, nanos.
547  * <p>
548  * This set can be used in conjunction with {@link #get(TemporalUnit)}
549  * to access the entire state of the duration.
550  *
551  * @return a list containing the seconds and nanos units, not null
552  */
553 @Override
554 public List<TemporalUnit> getUnits() {
555     return DurationUnits.UNITS;
556 }
557
558 /**
559  * Private class to delay initialization of this list until needed.
560  * The circular dependency between Duration and ChronoUnit prevents
561  * the simple initialization in Duration.
562  */
563 private static class DurationUnits {
564     static final List<TemporalUnit> UNITS =
565         Collections.unmodifiableList(Arrays.<TemporalUnit>asList(SECONDS, NANOS));
566 }
567
568 //-----
569 /**
570  * Checks if this duration is zero length.
571  * <p>
572  * A {@code Duration} represents a directed distance between two points on
573  * the time-line and can therefore be positive, zero or negative.
574  * This method checks whether the length is zero.
575  *
576  * @return true if this duration has a total length equal to zero
577  */
578 public boolean isZero() {
579     return (seconds | nanos) == 0;
580 }
581
582 /**
583  * Checks if this duration is negative, excluding zero.
584  * <p>
585  * A {@code Duration} represents a directed distance between two points on
586  * the time-line and can therefore be positive, zero or negative.
587  * This method checks whether the length is less than zero.
588  *
589  * @return true if this duration has a total length less than zero
590  */
591 public boolean isNegative() {
592     return seconds < 0;

```

```

593     }
594
595     //-----
596     /**
597      * Gets the number of seconds in this duration.
598      * <p>
599      * The length of the duration is stored using two fields - seconds and nanoseconds.
600      * The nanoseconds part is a value from 0 to 999,999,999 that is an adjustment to
601      * the length in seconds.
602      * The total duration is defined by calling this method and {@link #getNano()}.
603      * <p>
604      * A {@code Duration} represents a directed distance between two points on the time-line.
605      * A negative duration is expressed by the negative sign of the seconds part.
606      * A duration of -1 nanosecond is stored as -1 seconds plus 999,999,999 nanoseconds.
607      *
608      * @return the whole seconds part of the length of the duration, positive or negative
609      */
610     public long getSeconds() {
611         return seconds;
612     }
613
614     /**
615      * Gets the number of nanoseconds within the second in this duration.
616      * <p>
617      * The length of the duration is stored using two fields - seconds and nanoseconds.
618      * The nanoseconds part is a value from 0 to 999,999,999 that is an adjustment to
619      * the length in seconds.
620      * The total duration is defined by calling this method and {@link #getSeconds()}.
621      * <p>
622      * A {@code Duration} represents a directed distance between two points on the time-line.
623      * A negative duration is expressed by the negative sign of the seconds part.
624      * A duration of -1 nanosecond is stored as -1 seconds plus 999,999,999 nanoseconds.
625      *
626      * @return the nanoseconds within the second part of the length of the duration, from 0 to
627      *         999,999,999
628      */
629     public int getNano() {
630         return nanos;
631     }
632
633     //-----
634     /**
635      * Returns a copy of this duration with the specified amount of seconds.
636      * <p>
637      * This returns a duration with the specified seconds, retaining the
638      * nano-of-second part of this duration.
639      * <p>
640      * This instance is immutable and unaffected by this method call.
641      *
642      * @param seconds the seconds to represent, may be negative
643      * @return a {@code Duration} based on this period with the requested seconds, not null
644      */
645     public Duration withSeconds(long seconds) {

```

```

645     return create(seconds, nanos);
646 }
647
648 /**
649  * Returns a copy of this duration with the specified nano-of-second.
650  * <p>
651  * This returns a duration with the specified nano-of-second, retaining the
652  * seconds part of this duration.
653  * <p>
654  * This instance is immutable and unaffected by this method call.
655  *
656  * @param nanoOfSecond the nano-of-second to represent, from 0 to 999,999,999
657  * @return a {@code Duration} based on this period with the requested nano-of-second, not null
658  * @throws DateTimeException if the nano-of-second is invalid
659  */
660 public Duration withNanos(int nanoOfSecond) {
661     NANO_OF_SECOND.checkValidIntValue(nanoOfSecond);
662     return create(seconds, nanoOfSecond);
663 }
664
665 //-----
666 /**
667  * Returns a copy of this duration with the specified duration added.
668  * <p>
669  * This instance is immutable and unaffected by this method call.
670  *
671  * @param duration the duration to add, positive or negative, not null
672  * @return a {@code Duration} based on this duration with the specified duration added, not
673  *         null
674  * @throws ArithmeticException if numeric overflow occurs
675  */
676 public Duration plus(Duration duration) {
677     return plus(duration.getSeconds(), duration.getNano());
678 }
679
680 /**
681  * Returns a copy of this duration with the specified duration added.
682  * <p>
683  * The duration amount is measured in terms of the specified unit.
684  * Only a subset of units are accepted by this method.
685  * The unit must either have an {@link PlainTemporalUnit#isDurationEstimated() exact duration}
686  * or
687  * be {@link ChronoUnit#DAYS} which is treated as 24 hours. Other units throw an exception.
688  * <p>
689  * This instance is immutable and unaffected by this method call.
690  *
691  * @param amountToAdd the amount to add, measured in terms of the unit, positive or negative
692  * @param unit the unit that the amount is measured in, must have an exact duration, not null
693  * @return a {@code Duration} based on this duration with the specified duration added, not
694  *         null
695  * @throws UnsupportedTemporalTypeException if the unit is not supported
696  * @throws ArithmeticException if numeric overflow occurs
697  */

```



```

695 public Duration plus(long amountToAdd, TemporalUnit unit) {
696     Objects.requireNonNull(unit, "unit");
697     if (unit == DAYS) {
698         return plus(Math.multiplyExact(amountToAdd, SECONDS_PER_DAY), 0);
699     }
700     if (unit.isDurationEstimated()) {
701         throw new UnsupportedTemporalTypeException("Unit must not have an estimated duration");
702     }
703     if (amountToAdd == 0) {
704         return this;
705     }
706     if (unit instanceof ChronoUnit) {
707         switch ((ChronoUnit) unit) {
708             case NANOS: return plusNanos(amountToAdd);
709             case MICROS: return plusSeconds((amountToAdd / (1000_000L * 1000)) * 1000).
                plusNanos((amountToAdd % (1000_000L * 1000)) * 1000);
710             case MILLIS: return plusMillis(amountToAdd);
711             case SECONDS: return plusSeconds(amountToAdd);
712         }
713         return plusSeconds(Math.multiplyExact(unit.getDuration().seconds, amountToAdd));
714     }
715     Duration duration = unit.getDuration().multipliedBy(amountToAdd);
716     return plusSeconds(duration.getSeconds()).plusNanos(duration.getNano());
717 }
718
719 //-----
720 /**
721  * Returns a copy of this duration with the specified duration in standard 24 hour days added.
722  * <p>
723  * The number of days is multiplied by 86400 to obtain the number of seconds to add.
724  * This is based on the standard definition of a day as 24 hours.
725  * <p>
726  * This instance is immutable and unaffected by this method call.
727  *
728  * @param daysToAdd the days to add, positive or negative
729  * @return a {@code Duration} based on this duration with the specified days added, not null
730  * @throws ArithmeticException if numeric overflow occurs
731  */
732 public Duration plusDays(long daysToAdd) {
733     return plus(Math.multiplyExact(daysToAdd, SECONDS_PER_DAY), 0);
734 }
735
736 /**
737  * Returns a copy of this duration with the specified duration in hours added.
738  * <p>
739  * This instance is immutable and unaffected by this method call.
740  *
741  * @param hoursToAdd the hours to add, positive or negative
742  * @return a {@code Duration} based on this duration with the specified hours added, not null
743  * @throws ArithmeticException if numeric overflow occurs
744  */
745 public Duration plusHours(long hoursToAdd) {
746     return plus(Math.multiplyExact(hoursToAdd, SECONDS_PER_HOUR), 0);

```

```
747     }
748
749     /**
750      * Returns a copy of this duration with the specified duration in minutes added.
751      * <p>
752      * This instance is immutable and unaffected by this method call.
753      *
754      * @param minutesToAdd the minutes to add, positive or negative
755      * @return a {@code Duration} based on this duration with the specified minutes added, not null
756      * @throws ArithmeticException if numeric overflow occurs
757      */
758     public Duration plusMinutes(long minutesToAdd) {
759         return plus(Math.multiplyExact(minutesToAdd, SECONDS_PER_MINUTE), 0);
760     }
761
762     /**
763      * Returns a copy of this duration with the specified duration in seconds added.
764      * <p>
765      * This instance is immutable and unaffected by this method call.
766      *
767      * @param secondsToAdd the seconds to add, positive or negative
768      * @return a {@code Duration} based on this duration with the specified seconds added, not null
769      * @throws ArithmeticException if numeric overflow occurs
770      */
771     public Duration plusSeconds(long secondsToAdd) {
772         return plus(secondsToAdd, 0);
773     }
774
775     /**
776      * Returns a copy of this duration with the specified duration in milliseconds added.
777      * <p>
778      * This instance is immutable and unaffected by this method call.
779      *
780      * @param millisToAdd the milliseconds to add, positive or negative
781      * @return a {@code Duration} based on this duration with the specified milliseconds added, not
782      *         null
783      * @throws ArithmeticException if numeric overflow occurs
784      */
785     public Duration plusMillis(long millisToAdd) {
786         return plus(millisToAdd / 1000, (millisToAdd % 1000) * 1000_000);
787     }
788
789     /**
790      * Returns a copy of this duration with the specified duration in nanoseconds added.
791      * <p>
792      * This instance is immutable and unaffected by this method call.
793      *
794      * @param nanosToAdd the nanoseconds to add, positive or negative
795      * @return a {@code Duration} based on this duration with the specified nanoseconds added, not
796      *         null
797      * @throws ArithmeticException if numeric overflow occurs
798      */
799     public Duration plusNanos(long nanosToAdd) {
```

```

798         return plus(0, nanosToAdd);
799     }
800
801     /**
802      * Returns a copy of this duration with the specified duration added.
803      * <p>
804      * This instance is immutable and unaffected by this method call.
805      *
806      * @param secondsToAdd the seconds to add, positive or negative
807      * @param nanosToAdd the nanos to add, positive or negative
808      * @return a {@code Duration} based on this duration with the specified seconds added, not null
809      * @throws ArithmeticException if numeric overflow occurs
810      */
811     private Duration plus(long secondsToAdd, long nanosToAdd) {
812         if ((secondsToAdd | nanosToAdd) == 0) {
813             return this;
814         }
815         long epochSec = Math.addExact(seconds, secondsToAdd);
816         epochSec = Math.addExact(epochSec, nanosToAdd / NANOS_PER_SECOND);
817         nanosToAdd = nanosToAdd % NANOS_PER_SECOND;
818         long nanoAdjustment = nanos + nanosToAdd; // safe int+NANOS_PER_SECOND
819         return ofSeconds(epochSec, nanoAdjustment);
820     }
821
822     //-----
823     /**
824      * Returns a copy of this duration with the specified duration subtracted.
825      * <p>
826      * This instance is immutable and unaffected by this method call.
827      *
828      * @param duration the duration to subtract, positive or negative, not null
829      * @return a {@code Duration} based on this duration with the specified duration subtracted,
830      *         not null
831      * @throws ArithmeticException if numeric overflow occurs
832      */
833     public Duration minus(Duration duration) {
834         long secsToSubtract = duration.getSeconds();
835         int nanosToSubtract = duration.getNano();
836         if (secsToSubtract == Long.MIN_VALUE) {
837             return plus(Long.MAX_VALUE, -nanosToSubtract).plus(1, 0);
838         }
839         return plus(-secsToSubtract, -nanosToSubtract);
840     }
841
842     /**
843      * Returns a copy of this duration with the specified duration subtracted.
844      * <p>
845      * The duration amount is measured in terms of the specified unit.
846      * Only a subset of units are accepted by this method.
847      * The unit must either have an {@link TemporalUnit#isDurationEstimated() exact duration}
848      * or
849      * be {@link ChronoUnit#DAYS} which is treated as 24 hours. Other units throw an exception.
850      * <p>

```

```

849     * This instance is immutable and unaffected by this method call.
850     *
851     * @param amountToSubtract the amount to subtract, measured in terms of the unit, positive or
      negative
852     * @param unit the unit that the amount is measured in, must have an exact duration, not null
853     * @return a {@code Duration} based on this duration with the specified duration subtracted,
      not null
854     * @throws ArithmeticException if numeric overflow occurs
855     */
856     public Duration minus(long amountToSubtract, TemporalUnit unit) {
857         return (amountToSubtract == Long.MIN_VALUE ? plus(Long.MAX_VALUE, unit).plus(1, unit) :
      plus(-amountToSubtract, unit));
858     }
859
860     //-----
861     /**
862     * Returns a copy of this duration with the specified duration in standard 24 hour days
      subtracted.
863     * <p>
864     * The number of days is multiplied by 86400 to obtain the number of seconds to subtract.
865     * This is based on the standard definition of a day as 24 hours.
866     * <p>
867     * This instance is immutable and unaffected by this method call.
868     *
869     * @param daysToSubtract the days to subtract, positive or negative
870     * @return a {@code Duration} based on this duration with the specified days subtracted, not
      null
871     * @throws ArithmeticException if numeric overflow occurs
872     */
873     public Duration minusDays(long daysToSubtract) {
874         return (daysToSubtract == Long.MIN_VALUE ? plusDays(Long.MAX_VALUE).plusDays(1) : plusDays
      (-daysToSubtract));
875     }
876
877     /**
878     * Returns a copy of this duration with the specified duration in hours subtracted.
879     * <p>
880     * The number of hours is multiplied by 3600 to obtain the number of seconds to subtract.
881     * <p>
882     * This instance is immutable and unaffected by this method call.
883     *
884     * @param hoursToSubtract the hours to subtract, positive or negative
885     * @return a {@code Duration} based on this duration with the specified hours subtracted, not
      null
886     * @throws ArithmeticException if numeric overflow occurs
887     */
888     public Duration minusHours(long hoursToSubtract) {
889         return (hoursToSubtract == Long.MIN_VALUE ? plusHours(Long.MAX_VALUE).plusHours(1) :
      plusHours(-hoursToSubtract));
890     }
891
892     /**
893     * Returns a copy of this duration with the specified duration in minutes subtracted.

```

```
894     * <p>
895     * The number of hours is multiplied by 60 to obtain the number of seconds to subtract.
896     * <p>
897     * This instance is immutable and unaffected by this method call.
898     *
899     * @param minutesToSubtract the minutes to subtract, positive or negative
900     * @return a {@code Duration} based on this duration with the specified minutes subtracted, not
901     *         null
902     * @throws ArithmeticException if numeric overflow occurs
903     */
904     public Duration minusMinutes(long minutesToSubtract) {
905         return (minutesToSubtract == Long.MIN_VALUE ? plusMinutes(Long.MAX_VALUE).plusMinutes(1) :
906             plusMinutes(-minutesToSubtract));
907     }
908
909     /**
910     * Returns a copy of this duration with the specified duration in seconds subtracted.
911     * <p>
912     * This instance is immutable and unaffected by this method call.
913     *
914     * @param secondsToSubtract the seconds to subtract, positive or negative
915     * @return a {@code Duration} based on this duration with the specified seconds subtracted, not
916     *         null
917     * @throws ArithmeticException if numeric overflow occurs
918     */
919     public Duration minusSeconds(long secondsToSubtract) {
920         return (secondsToSubtract == Long.MIN_VALUE ? plusSeconds(Long.MAX_VALUE).plusSeconds(1) :
921             plusSeconds(-secondsToSubtract));
922     }
923
924     /**
925     * Returns a copy of this duration with the specified duration in milliseconds subtracted.
926     * <p>
927     * This instance is immutable and unaffected by this method call.
928     *
929     * @param millisToSubtract the milliseconds to subtract, positive or negative
930     * @return a {@code Duration} based on this duration with the specified milliseconds subtracted
931     *         , not null
932     * @throws ArithmeticException if numeric overflow occurs
933     */
934     public Duration minusMillis(long millisToSubtract) {
935         return (millisToSubtract == Long.MIN_VALUE ? plusMillis(Long.MAX_VALUE).plusMillis(1) :
936             plusMillis(-millisToSubtract));
937     }
938
939     /**
940     * Returns a copy of this duration with the specified duration in nanoseconds subtracted.
941     * <p>
942     * This instance is immutable and unaffected by this method call.
943     *
944     * @param nanosToSubtract the nanoseconds to subtract, positive or negative
945     * @return a {@code Duration} based on this duration with the specified nanoseconds subtracted,
946     *         not null
```

```

940     * @throws ArithmeticException if numeric overflow occurs
941     */
942     public Duration minusNanos(long nanosToSubtract) {
943         return (nanosToSubtract == Long.MIN_VALUE ? plusNanos(Long.MAX_VALUE).plusNanos(1) :
944             plusNanos(-nanosToSubtract));
945     }
946
947     //-----
948     /**
949     * Returns a copy of this duration multiplied by the scalar.
950     * <p>
951     * This instance is immutable and unaffected by this method call.
952     *
953     * @param multiplicand the value to multiply the duration by, positive or negative
954     * @return a {@code Duration} based on this duration multiplied by the specified scalar, not
955     *         null
956     * @throws ArithmeticException if numeric overflow occurs
957     */
958     public Duration multipliedBy(long multiplicand) {
959         if (multiplicand == 0) {
960             return ZERO;
961         }
962         if (multiplicand == 1) {
963             return this;
964         }
965         return create(toSeconds().multiply(BigDecimal.valueOf(multiplicand)));
966     }
967
968     /**
969     * Returns a copy of this duration divided by the specified value.
970     * <p>
971     * This instance is immutable and unaffected by this method call.
972     *
973     * @param divisor the value to divide the duration by, positive or negative, not zero
974     * @return a {@code Duration} based on this duration divided by the specified divisor, not null
975     * @throws ArithmeticException if the divisor is zero or if numeric overflow occurs
976     */
977     public Duration dividedBy(long divisor) {
978         if (divisor == 0) {
979             throw new ArithmeticException("Cannot divide by zero");
980         }
981         if (divisor == 1) {
982             return this;
983         }
984         return create(toSeconds().divide(BigDecimal.valueOf(divisor), RoundingMode.DOWN));
985     }
986
987     /**
988     * Converts this duration to the total length in seconds and
989     * fractional nanoseconds expressed as a {@code BigDecimal}.
990     *
991     * @return the total length of the duration in seconds, with a scale of 9, not null
992     */

```

```

991 private BigDecimal toSeconds() {
992     return BigDecimal.valueOf(seconds).add(BigDecimal.valueOf(nanos, 9));
993 }
994
995 /**
996  * Creates an instance of {@code Duration} from a number of seconds.
997  *
998  * @param seconds the number of seconds, up to scale 9, positive or negative
999  * @return a {@code Duration}, not null
1000  * @throws ArithmeticException if numeric overflow occurs
1001  */
1002 private static Duration create(BigDecimal seconds) {
1003     BigInteger nanos = seconds.movePointRight(9).toBigIntegerExact();
1004     BigInteger[] divRem = nanos.divideAndRemainder(BI_NANOS_PER_SECOND);
1005     if (divRem[0].bitLength() > 63) {
1006         throw new ArithmeticException("Exceeds capacity of Duration: " + nanos);
1007     }
1008     return ofSeconds(divRem[0].longValue(), divRem[1].intValue());
1009 }
1010
1011 //-----
1012 /**
1013  * Returns a copy of this duration with the length negated.
1014  * <p>
1015  * This method swaps the sign of the total length of this duration.
1016  * For example, {@code PT1.3S} will be returned as {@code PT-1.3S}.
1017  * <p>
1018  * This instance is immutable and unaffected by this method call.
1019  *
1020  * @return a {@code Duration} based on this duration with the amount negated, not null
1021  * @throws ArithmeticException if numeric overflow occurs
1022  */
1023 public Duration negated() {
1024     return multipliedBy(-1);
1025 }
1026
1027 /**
1028  * Returns a copy of this duration with a positive length.
1029  * <p>
1030  * This method returns a positive duration by effectively removing the sign from any negative
1031  * total length.
1032  * For example, {@code PT-1.3S} will be returned as {@code PT1.3S}.
1033  * <p>
1034  * This instance is immutable and unaffected by this method call.
1035  *
1036  * @return a {@code Duration} based on this duration with an absolute length, not null
1037  * @throws ArithmeticException if numeric overflow occurs
1038  */
1039 public Duration abs() {
1040     return isNegative() ? negated() : this;
1041 }
1042 //-----

```

```

1043 /**
1044  * Adds this duration to the specified temporal object.
1045  * <p>
1046  * This returns a temporal object of the same observable type as the input
1047  * with this duration added.
1048  * <p>
1049  * In most cases, it is clearer to reverse the calling pattern by using
1050  * {@link Temporal#plus(TemporalAmount)}.
1051  * <pre>
1052  * // these two lines are equivalent, but the second approach is recommended
1053  * dateTime = thisDuration.addTo(dateTime);
1054  * dateTime = dateTime.plus(thisDuration);
1055  * </pre>
1056  * <p>
1057  * The calculation will add the seconds, then nanos.
1058  * Only non-zero amounts will be added.
1059  * <p>
1060  * This instance is immutable and unaffected by this method call.
1061  *
1062  * @param temporal the temporal object to adjust, not null
1063  * @return an object of the same type with the adjustment made, not null
1064  * @throws DateTimeException if unable to add
1065  * @throws ArithmeticException if numeric overflow occurs
1066  */
1067 @Override
1068 public Temporal addTo(Temporal temporal) {
1069     if (seconds != 0) {
1070         temporal = temporal.plus(seconds, SECONDS);
1071     }
1072     if (nanos != 0) {
1073         temporal = temporal.plus(nanos, NANOS);
1074     }
1075     return temporal;
1076 }
1077
1078 /**
1079  * Subtracts this duration from the specified temporal object.
1080  * <p>
1081  * This returns a temporal object of the same observable type as the input
1082  * with this duration subtracted.
1083  * <p>
1084  * In most cases, it is clearer to reverse the calling pattern by using
1085  * {@link Temporal#minus(TemporalAmount)}.
1086  * <pre>
1087  * // these two lines are equivalent, but the second approach is recommended
1088  * dateTime = thisDuration.subtractFrom(dateTime);
1089  * dateTime = dateTime.minus(thisDuration);
1090  * </pre>
1091  * <p>
1092  * The calculation will subtract the seconds, then nanos.
1093  * Only non-zero amounts will be added.
1094  * <p>
1095  * This instance is immutable and unaffected by this method call.

```



```
1096 *
1097 * @param temporal the temporal object to adjust, not null
1098 * @return an object of the same type with the adjustment made, not null
1099 * @throws DateTimeException if unable to subtract
1100 * @throws ArithmeticException if numeric overflow occurs
1101 */
1102 @Override
1103 public Temporal subtractFrom(Temporal temporal) {
1104     if (seconds != 0) {
1105         temporal = temporal.minus(seconds, SECONDS);
1106     }
1107     if (nanos != 0) {
1108         temporal = temporal.minus(nanos, NANOS);
1109     }
1110     return temporal;
1111 }
1112
1113 //-----
1114 /**
1115  * Gets the number of days in this duration.
1116  * <p>
1117  * This returns the total number of days in the duration by dividing the
1118  * number of seconds by 86400.
1119  * This is based on the standard definition of a day as 24 hours.
1120  * <p>
1121  * This instance is immutable and unaffected by this method call.
1122  *
1123  * @return the number of days in the duration, may be negative
1124  */
1125 public long toDays() {
1126     return seconds / SECONDS_PER_DAY;
1127 }
1128
1129 /**
1130  * Gets the number of hours in this duration.
1131  * <p>
1132  * This returns the total number of hours in the duration by dividing the
1133  * number of seconds by 3600.
1134  * <p>
1135  * This instance is immutable and unaffected by this method call.
1136  *
1137  * @return the number of hours in the duration, may be negative
1138  */
1139 public long toHours() {
1140     return seconds / SECONDS_PER_HOUR;
1141 }
1142
1143 /**
1144  * Gets the number of minutes in this duration.
1145  * <p>
1146  * This returns the total number of minutes in the duration by dividing the
1147  * number of seconds by 60.
1148  * <p>
```

```

1149     * This instance is immutable and unaffected by this method call.
1150     *
1151     * @return the number of minutes in the duration, may be negative
1152     */
1153     public long toMinutes() {
1154         return seconds / SECONDS_PER_MINUTE;
1155     }
1156
1157     /**
1158     * Converts this duration to the total length in milliseconds.
1159     * <p>
1160     * If this duration is too large to fit in a {@code long} milliseconds, then an
1161     * exception is thrown.
1162     * <p>
1163     * If this duration has greater than millisecond precision, then the conversion
1164     * will drop any excess precision information as though the amount in nanoseconds
1165     * was subject to integer division by one million.
1166     *
1167     * @return the total length of the duration in milliseconds
1168     * @throws ArithmeticException if numeric overflow occurs
1169     */
1170     public long toMillis() {
1171         long millis = Math.multiplyExact(seconds, 1000);
1172         millis = Math.addExact(millis, nanos / 1000_000);
1173         return millis;
1174     }
1175
1176     /**
1177     * Converts this duration to the total length in nanoseconds expressed as a {@code long}.
1178     * <p>
1179     * If this duration is too large to fit in a {@code long} nanoseconds, then an
1180     * exception is thrown.
1181     *
1182     * @return the total length of the duration in nanoseconds
1183     * @throws ArithmeticException if numeric overflow occurs
1184     */
1185     public long toNanos() {
1186         long totalNanos = Math.multiplyExact(seconds, NANOS_PER_SECOND);
1187         totalNanos = Math.addExact(totalNanos, nanos);
1188         return totalNanos;
1189     }
1190
1191     //-----
1192     /**
1193     * Compares this duration to the specified {@code Duration}.
1194     * <p>
1195     * The comparison is based on the total length of the durations.
1196     * It is "consistent with equals", as defined by {@link Comparable}.
1197     *
1198     * @param otherDuration the other duration to compare to, not null
1199     * @return the comparator value, negative if less, positive if greater
1200     */
1201     @Override

```

```

1202     public int compareTo(Duration otherDuration) {
1203         int cmp = Long.compare(seconds, otherDuration.seconds);
1204         if (cmp != 0) {
1205             return cmp;
1206         }
1207         return nanos - otherDuration.nanos;
1208     }
1209
1210     //-----
1211     /**
1212      * Checks if this duration is equal to the specified {@code Duration}.
1213      * <p>
1214      * The comparison is based on the total length of the durations.
1215      *
1216      * @param otherDuration the other duration, null returns false
1217      * @return true if the other duration is equal to this one
1218      */
1219     @Override
1220     public boolean equals(Object otherDuration) {
1221         if (this == otherDuration) {
1222             return true;
1223         }
1224         if (otherDuration instanceof Duration) {
1225             Duration other = (Duration) otherDuration;
1226             return this.seconds == other.seconds &&
1227                    this.nanos == other.nanos;
1228         }
1229         return false;
1230     }
1231
1232     /**
1233      * A hash code for this duration.
1234      *
1235      * @return a suitable hash code
1236      */
1237     @Override
1238     public int hashCode() {
1239         return ((int) (seconds ^ (seconds >>> 32))) + (51 * nanos);
1240     }
1241
1242     //-----
1243     /**
1244      * A string representation of this duration using ISO-8601 seconds
1245      * based representation, such as {@code PT8H6M12.345S}.
1246      * <p>
1247      * The format of the returned string will be {@code PTnHnMnS}, where n is
1248      * the relevant hours, minutes or seconds part of the duration.
1249      * Any fractional seconds are placed after a decimal point i the seconds section.
1250      * If a section has a zero value, it is omitted.
1251      * The hours, minutes and seconds will all have the same sign.
1252      * <p>
1253      * Examples:
1254      * <pre>

```

```

1255 * "20.345 seconds" -- "PT20.345S
1256 * "15 minutes" (15 * 60 seconds) -- "PT15M"
1257 * "10 hours" (10 * 3600 seconds) -- "PT10H"
1258 * "2 days" (2 * 86400 seconds) -- "PT48H"
1259 * </pre>
1260 * Note that multiples of 24 hours are not output as days to avoid confusion
1261 * with {@code Period}.
1262 *
1263 * @return an ISO-8601 representation of this duration, not null
1264 */
1265 @Override
1266 public String toString() {
1267     if (this == ZERO) {
1268         return "PT0S";
1269     }
1270     long hours = seconds / SECONDS_PER_HOUR;
1271     int minutes = (int) ((seconds % SECONDS_PER_HOUR) / SECONDS_PER_MINUTE);
1272     int secs = (int) (seconds % SECONDS_PER_MINUTE);
1273     StringBuilder buf = new StringBuilder(24);
1274     buf.append("PT");
1275     if (hours != 0) {
1276         buf.append(hours).append('H');
1277     }
1278     if (minutes != 0) {
1279         buf.append(minutes).append('M');
1280     }
1281     if (secs == 0 && nanos == 0 && buf.length() > 2) {
1282         return buf.toString();
1283     }
1284     if (secs < 0 && nanos > 0) {
1285         if (secs == -1) {
1286             buf.append("-0");
1287         } else {
1288             buf.append(secs + 1);
1289         }
1290     } else {
1291         buf.append(secs);
1292     }
1293     if (nanos > 0) {
1294         int pos = buf.length();
1295         if (secs < 0) {
1296             buf.append(2 * NANOS_PER_SECOND - nanos);
1297         } else {
1298             buf.append(nanos + NANOS_PER_SECOND);
1299         }
1300         while (buf.charAt(buf.length() - 1) == '0') {
1301             buf.setLength(buf.length() - 1);
1302         }
1303         buf.setCharAt(pos, '.');
1304     }
1305     buf.append('S');
1306     return buf.toString();
1307 }

```

```

1308
1309 //-----
1310 /**
1311  * Writes the object using a
1312  * <a href="../../serialized-form.html#java.time.Ser">dedicated serialized form</a>.
1313  * @serialData
1314  * <pre>
1315  *   out.writeByte(1); // identifies a Duration
1316  *   out.writeLong(seconds);
1317  *   out.writeInt(nanos);
1318  * </pre>
1319  *
1320  * @return the instance of {@code Ser}, not null
1321  */
1322 private Object writeReplace() {
1323     return new Ser(Ser.DURATION_TYPE, this);
1324 }
1325
1326 /**
1327  * Defend against malicious streams.
1328  *
1329  * @param s the stream to read
1330  * @throws InvalidObjectException always
1331  */
1332 private void readObject(ObjectInputStream s) throws InvalidObjectException {
1333     throw new InvalidObjectException("Deserialization via serialization delegate");
1334 }
1335
1336 void writeExternal(DataOutput out) throws IOException {
1337     out.writeLong(seconds);
1338     out.writeInt(nanos);
1339 }
1340
1341 static Duration readExternal(DataInput in) throws IOException {
1342     long seconds = in.readLong();
1343     int nanos = in.readInt();
1344     return Duration.ofSeconds(seconds, nanos);
1345 }
1346
1347 }

```

4.5 Instant.java

Secuencia 54: java/time/Instant.java

```

1 /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *

```

```
10  *
11  *
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26 /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62 package java.time;
```

```
63
64 import static java.time.LocalDateTime.NANOS_PER_SECOND;
65 import static java.time.LocalDateTime.SECONDS_PER_DAY;
66 import static java.time.LocalDateTime.SECONDS_PER_HOUR;
67 import static java.time.LocalDateTime.SECONDS_PER_MINUTE;
68 import static java.time.temporal.ChronoField.INSTANT_SECONDS;
69 import static java.time.temporal.ChronoField.MICRO_OF_SECOND;
70 import static java.time.temporal.ChronoField.MILLI_OF_SECOND;
71 import static java.time.temporal.ChronoField.NANO_OF_SECOND;
72 import static java.time.temporal.ChronoUnit.DAYS;
73 import static java.time.temporal.ChronoUnit.NANOS;
74
75 import java.io.DataInput;
76 import java.io.DataOutput;
77 import java.io.IOException;
78 import java.io.InvalidObjectException;
79 import java.io.ObjectInputStream;
80 import java.io.Serializable;
81 import java.time.format.DateTimeFormatter;
82 import java.time.format.DateTimeParseException;
83 import java.time.temporal.ChronoField;
84 import java.time.temporal.ChronoUnit;
85 import java.time.temporal.Temporal;
86 import java.time.temporal.TemporalAccessor;
87 import java.time.temporal.TemporalAdjuster;
88 import java.time.temporal.TemporalAmount;
89 import java.time.temporal.TemporalField;
90 import java.time.temporal.TemporalQueries;
91 import java.time.temporal.TemporalQuery;
92 import java.time.temporal.TemporalUnit;
93 import java.time.temporal.UnsupportedTemporalTypeException;
94 import java.time.temporal.ValueRange;
95 import java.util.Objects;
96
97 /**
98  * An instantaneous point on the time-line.
99  * <p>
100  * This class models a single instantaneous point on the time-line.
101  * This might be used to record event time-stamps in the application.
102  * <p>
103  * The range of an instant requires the storage of a number larger than a {@code long}.
104  * To achieve this, the class stores a {@code long} representing epoch-seconds and an
105  * {@code int} representing nanosecond-of-second, which will always be between 0 and 999,999,999.
106  * The epoch-seconds are measured from the standard Java epoch of {@code 1970-01-01T00:00:00Z}
107  * where instants after the epoch have positive values, and earlier instants have negative values.
108  * For both the epoch-second and nanosecond parts, a larger value is always later on the time-line
109  * than a smaller value.
110  *
111  * <h3>Time-scale</h3>
112  * <p>
113  * The length of the solar day is the standard way that humans measure time.
114  * This has traditionally been subdivided into 24 hours of 60 minutes of 60 seconds,
115  * forming a 86400 second day.
```

```
116 * <p>
117 * Modern timekeeping is based on atomic clocks which precisely define an SI second
118 * relative to the transitions of a Caesium atom. The length of an SI second was defined
119 * to be very close to the 86400th fraction of a day.
120 * <p>
121 * Unfortunately, as the Earth rotates the length of the day varies.
122 * In addition, over time the average length of the day is getting longer as the Earth slows.
123 * As a result, the length of a solar day in 2012 is slightly longer than 86400 SI seconds.
124 * The actual length of any given day and the amount by which the Earth is slowing
125 * are not predictable and can only be determined by measurement.
126 * The UT1 time-scale captures the accurate length of day, but is only available some
127 * time after the day has completed.
128 * <p>
129 * The UTC time-scale is a standard approach to bundle up all the additional fractions
130 * of a second from UT1 into whole seconds, known as <i>leap-seconds</i>.
131 * A leap-second may be added or removed depending on the Earth's rotational changes.
132 * As such, UTC permits a day to have 86399 SI seconds or 86401 SI seconds where
133 * necessary in order to keep the day aligned with the Sun.
134 * <p>
135 * The modern UTC time-scale was introduced in 1972, introducing the concept of whole leap-seconds.
136 * Between 1958 and 1972, the definition of UTC was complex, with minor sub-second leaps and
137 * alterations to the length of the notional second. As of 2012, discussions are underway
138 * to change the definition of UTC again, with the potential to remove leap seconds or
139 * introduce other changes.
140 * <p>
141 * Given the complexity of accurate timekeeping described above, this Java API defines
142 * its own time-scale, the <i>Java Time-Scale</i>.
143 * <p>
144 * The Java Time-Scale divides each calendar day into exactly 86400
145 * subdivisions, known as seconds. These seconds may differ from the
146 * SI second. It closely matches the de facto international civil time
147 * scale, the definition of which changes from time to time.
148 * <p>
149 * The Java Time-Scale has slightly different definitions for different
150 * segments of the time-line, each based on the consensus international
151 * time scale that is used as the basis for civil time. Whenever the
152 * internationally-agreed time scale is modified or replaced, a new
153 * segment of the Java Time-Scale must be defined for it. Each segment
154 * must meet these requirements:
155 * <ul>
156 * <li>the Java Time-Scale shall closely match the underlying international
157 * civil time scale;</li>
158 * <li>the Java Time-Scale shall exactly match the international civil
159 * time scale at noon each day;</li>
160 * <li>the Java Time-Scale shall have a precisely-defined relationship to
161 * the international civil time scale.</li>
162 * </ul>
163 * There are currently, as of 2013, two segments in the Java time-scale.
164 * <p>
165 * For the segment from 1972-11-03 (exact boundary discussed below) until
166 * further notice, the consensus international time scale is UTC (with
167 * leap seconds). In this segment, the Java Time-Scale is identical to
168 * <a href="http://www.cl.cam.ac.uk/~mgk25/time/utc-sls/">UTC-SLS</a>.
```



```

169 * This is identical to UTC on days that do not have a leap second.
170 * On days that do have a leap second, the leap second is spread equally
171 * over the last 1000 seconds of the day, maintaining the appearance of
172 * exactly 86400 seconds per day.
173 * <p>
174 * For the segment prior to 1972-11-03, extending back arbitrarily far,
175 * the consensus international time scale is defined to be UT1, applied
176 * proleptically, which is equivalent to the (mean) solar time on the
177 * prime meridian (Greenwich). In this segment, the Java Time-Scale is
178 * identical to the consensus international time scale. The exact
179 * boundary between the two segments is the instant where UT1 = UTC
180 * between 1972-11-03T00:00 and 1972-11-04T12:00.
181 * <p>
182 * Implementations of the Java time-scale using the JSR-310 API are not
183 * required to provide any clock that is sub-second accurate, or that
184 * progresses monotonically or smoothly. Implementations are therefore
185 * not required to actually perform the UTC-SLS slew or to otherwise be
186 * aware of leap seconds. JSR-310 does, however, require that
187 * implementations must document the approach they use when defining a
188 * clock representing the current instant.
189 * See {@link Clock} for details on the available clocks.
190 * <p>
191 * The Java time-scale is used for all date-time classes.
192 * This includes {@code Instant}, {@code LocalDate}, {@code LocalDateTime}, {@code OffsetDateTime},
193 * {@code ZonedDateTime} and {@code Duration}.
194 *
195 * <p>
196 * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
197 * class; use of identity-sensitive operations (including reference equality
198 * ({@code ==}), identity hash code, or synchronization) on instances of
199 * {@code Instant} may have unpredictable results and should be avoided.
200 * The {@code equals} method should be used for comparisons.
201 *
202 * @implSpec
203 * This class is immutable and thread-safe.
204 *
205 * @since 1.8
206 */
207 public final class Instant
208     implements Temporal, TemporalAdjuster, Comparable<Instant>, Serializable {
209
210     /**
211      * Constant for the 1970-01-01T00:00:00Z epoch instant.
212      */
213     public static final Instant EPOCH = new Instant(0, 0);
214     /**
215      * The minimum supported epoch second.
216      */
217     private static final long MIN_SECOND = -31557014167219200L;
218     /**
219      * The maximum supported epoch second.
220      */
221     private static final long MAX_SECOND = 31556889864403199L;

```

```

222  /**
223   * The minimum supported {@code Instant}, '-1000000000-01-01T00:00Z'.
224   * This could be used by an application as a "far past" instant.
225   * <p>
226   * This is one year earlier than the minimum {@code LocalDateTime}.
227   * This provides sufficient values to handle the range of {@code ZoneOffset}
228   * which affect the instant in addition to the local date-time.
229   * The value is also chosen such that the value of the year fits in
230   * an {@code int}.
231   */
232  public static final Instant MIN = Instant.ofEpochSecond(MIN_SECOND, 0);
233  /**
234   * The maximum supported {@code Instant}, '1000000000-12-31T23:59:59.999999999Z'.
235   * This could be used by an application as a "far future" instant.
236   * <p>
237   * This is one year later than the maximum {@code LocalDateTime}.
238   * This provides sufficient values to handle the range of {@code ZoneOffset}
239   * which affect the instant in addition to the local date-time.
240   * The value is also chosen such that the value of the year fits in
241   * an {@code int}.
242   */
243  public static final Instant MAX = Instant.ofEpochSecond(MAX_SECOND, 999_999_999);
244
245  /**
246   * Serialization version.
247   */
248  private static final long serialVersionUID = -665713676816604388L;
249
250  /**
251   * The number of seconds from the epoch of 1970-01-01T00:00:00Z.
252   */
253  private final long seconds;
254  /**
255   * The number of nanoseconds, later along the time-line, from the seconds field.
256   * This is always positive, and never exceeds 999,999,999.
257   */
258  private final int nanos;
259
260  //-----
261  /**
262   * Obtains the current instant from the system clock.
263   * <p>
264   * This will query the {@link Clock#systemUTC() system UTC clock} to
265   * obtain the current instant.
266   * <p>
267   * Using this method will prevent the ability to use an alternate time-source for
268   * testing because the clock is effectively hard-coded.
269   *
270   * @return the current instant using the system clock, not null
271   */
272  public static Instant now() {
273      return Clock.systemUTC().instant();
274  }

```

```

275
276 /**
277  * Obtains the current instant from the specified clock.
278  * <p>
279  * This will query the specified clock to obtain the current time.
280  * <p>
281  * Using this method allows the use of an alternate clock for testing.
282  * The alternate clock may be introduced using {@link Clock dependency injection}.
283  *
284  * @param clock the clock to use, not null
285  * @return the current instant, not null
286  */
287 public static Instant now(Clock clock) {
288     Objects.requireNonNull(clock, "clock");
289     return clock.instant();
290 }
291
292 //-----
293 /**
294  * Obtains an instance of {@code Instant} using seconds from the
295  * epoch of 1970-01-01T00:00:00Z.
296  * <p>
297  * The nanosecond field is set to zero.
298  *
299  * @param epochSecond the number of seconds from 1970-01-01T00:00:00Z
300  * @return an instant, not null
301  * @throws DateTimeException if the instant exceeds the maximum or minimum instant
302  */
303 public static Instant ofEpochSecond(long epochSecond) {
304     return create(epochSecond, 0);
305 }
306
307 /**
308  * Obtains an instance of {@code Instant} using seconds from the
309  * epoch of 1970-01-01T00:00:00Z and nanosecond fraction of second.
310  * <p>
311  * This method allows an arbitrary number of nanoseconds to be passed in.
312  * The factory will alter the values of the second and nanosecond in order
313  * to ensure that the stored nanosecond is in the range 0 to 999,999,999.
314  * For example, the following will result in the exactly the same instant:
315  * <pre>
316  * Instant.ofEpochSecond(3, 1);
317  * Instant.ofEpochSecond(4, -999_999_999);
318  * Instant.ofEpochSecond(2, 1000_000_001);
319  * </pre>
320  *
321  * @param epochSecond the number of seconds from 1970-01-01T00:00:00Z
322  * @param nanoAdjustment the nanosecond adjustment to the number of seconds, positive or
323     negative
324  * @return an instant, not null
325  * @throws DateTimeException if the instant exceeds the maximum or minimum instant
326  * @throws ArithmeticException if numeric overflow occurs
327  */

```

```

327 public static Instant ofEpochSecond(long epochSecond, long nanoAdjustment) {
328     long secs = Math.addExact(epochSecond, Math.floorDiv(nanoAdjustment, NANOS_PER_SECOND));
329     int nos = (int) Math.floorMod(nanoAdjustment, NANOS_PER_SECOND);
330     return create(secs, nos);
331 }
332
333 /**
334  * Obtains an instance of {@code Instant} using milliseconds from the
335  * epoch of 1970-01-01T00:00:00Z.
336  * <p>
337  * The seconds and nanoseconds are extracted from the specified milliseconds.
338  *
339  * @param epochMilli the number of milliseconds from 1970-01-01T00:00:00Z
340  * @return an instant, not null
341  * @throws DateTimeException if the instant exceeds the maximum or minimum instant
342  */
343 public static Instant ofEpochMilli(long epochMilli) {
344     long secs = Math.floorDiv(epochMilli, 1000);
345     int mos = (int) Math.floorMod(epochMilli, 1000);
346     return create(secs, mos * 1000_000);
347 }
348
349 //-----
350 /**
351  * Obtains an instance of {@code Instant} from a temporal object.
352  * <p>
353  * This obtains an instant based on the specified temporal.
354  * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
355  * which this factory converts to an instance of {@code Instant}.
356  * <p>
357  * The conversion extracts the {@link ChronoField#INSTANT_SECONDS INSTANT_SECONDS}
358  * and {@link ChronoField#NANO_OF_SECOND NANO_OF_SECOND} fields.
359  * <p>
360  * This method matches the signature of the functional interface {@link TemporalQuery}
361  * allowing it to be used as a query via method reference, {@code Instant::from}.
362  *
363  * @param temporal the temporal object to convert, not null
364  * @return the instant, not null
365  * @throws DateTimeException if unable to convert to an {@code Instant}
366  */
367 public static Instant from(TemporalAccessor temporal) {
368     if (temporal instanceof Instant) {
369         return (Instant) temporal;
370     }
371     Objects.requireNonNull(temporal, "temporal");
372     try {
373         long instantSecs = temporal.getLong(INSTANT_SECONDS);
374         int nanoOfSecond = temporal.get(NANO_OF_SECOND);
375         return Instant.ofEpochSecond(instantSecs, nanoOfSecond);
376     } catch (DateTimeException ex) {
377         throw new DateTimeException("Unable to obtain Instant from TemporalAccessor: " +
378             temporal + " of type " + temporal.getClass().getName(), ex);
379     }

```

```

380     }
381
382     //-----
383     /**
384      * Obtains an instance of {@code Instant} from a text string such as
385      * {@code 2007-12-03T10:15:30.00Z}.
386      * <p>
387      * The string must represent a valid instant in UTC and is parsed using
388      * {@link DateTimeFormatter#ISO_INSTANT}.
389      *
390      * @param text the text to parse, not null
391      * @return the parsed instant, not null
392      * @throws DateTimeParseException if the text cannot be parsed
393      */
394     public static Instant parse(final CharSequence text) {
395         return DateTimeFormatter.ISO_INSTANT.parse(text, Instant::from);
396     }
397
398     //-----
399     /**
400      * Obtains an instance of {@code Instant} using seconds and nanoseconds.
401      *
402      * @param seconds the length of the duration in seconds
403      * @param nanoOfSecond the nano-of-second, from 0 to 999,999,999
404      * @throws DateTimeException if the instant exceeds the maximum or minimum instant
405      */
406     private static Instant create(long seconds, int nanoOfSecond) {
407         if ((seconds | nanoOfSecond) == 0) {
408             return EPOCH;
409         }
410         if (seconds < MIN_SECOND || seconds > MAX_SECOND) {
411             throw new DateTimeException("Instant exceeds minimum or maximum instant");
412         }
413         return new Instant(seconds, nanoOfSecond);
414     }
415
416     /**
417      * Constructs an instance of {@code Instant} using seconds from the epoch of
418      * 1970-01-01T00:00:00Z and nanosecond fraction of second.
419      *
420      * @param epochSecond the number of seconds from 1970-01-01T00:00:00Z
421      * @param nanos the nanoseconds within the second, must be positive
422      */
423     private Instant(long epochSecond, int nanos) {
424         super();
425         this.seconds = epochSecond;
426         this.nanos = nanos;
427     }
428
429     //-----
430     /**
431      * Checks if the specified field is supported.
432      * <p>

```

```

433 * This checks if this instant can be queried for the specified field.
434 * If false, then calling the {@link #range(TemporalField) range},
435 * {@link #get(TemporalField) get} and {@link #with(TemporalField, long)}
436 * methods will throw an exception.
437 * <p>
438 * If the field is a {@link ChronoField} then the query is implemented here.
439 * The supported fields are:
440 * <ul>
441 * <li>{@code NANO_OF_SECOND}
442 * <li>{@code MICRO_OF_SECOND}
443 * <li>{@code MILLI_OF_SECOND}
444 * <li>{@code INSTANT_SECONDS}
445 * </ul>
446 * All other {@code ChronoField} instances will return false.
447 * <p>
448 * If the field is not a {@code ChronoField}, then the result of this method
449 * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
450 * passing {@code this} as the argument.
451 * Whether the field is supported is determined by the field.
452 *
453 * @param field the field to check, null returns false
454 * @return true if the field is supported on this instant, false if not
455 */
456 @Override
457 public boolean isSupported(TemporalField field) {
458     if (field instanceof ChronoField) {
459         return field == INSTANT_SECONDS || field == NANO_OF_SECOND || field == MICRO_OF_SECOND
460             || field == MILLI_OF_SECOND;
461     }
462     return field != null && field.isSupportedBy(this);
463 }
464 /**
465 * Checks if the specified unit is supported.
466 * <p>
467 * This checks if the specified unit can be added to, or subtracted from, this date-time.
468 * If false, then calling the {@link #plus(long, TemporalUnit)} and
469 * {@link #minus(long, TemporalUnit) minus} methods will throw an exception.
470 * <p>
471 * If the unit is a {@link ChronoUnit} then the query is implemented here.
472 * The supported units are:
473 * <ul>
474 * <li>{@code NANOS}
475 * <li>{@code MICROS}
476 * <li>{@code MILLIS}
477 * <li>{@code SECONDS}
478 * <li>{@code MINUTES}
479 * <li>{@code HOURS}
480 * <li>{@code HALF_DAYS}
481 * <li>{@code DAYS}
482 * </ul>
483 * All other {@code ChronoUnit} instances will return false.
484 * <p>

```

```

485     * If the unit is not a {@code ChronoUnit}, then the result of this method
486     * is obtained by invoking {@code TemporalUnit.isSupportedBy(Temporal)}
487     * passing {@code this} as the argument.
488     * Whether the unit is supported is determined by the unit.
489     *
490     * @param unit the unit to check, null returns false
491     * @return true if the unit can be added/subtracted, false if not
492     */
493     @Override
494     public boolean isSupported(TemporalUnit unit) {
495         if (unit instanceof ChronoUnit) {
496             return unit.isTimeBased() || unit == DAYS;
497         }
498         return unit != null && unit.isSupportedBy(this);
499     }
500
501     //-----
502     /**
503     * Gets the range of valid values for the specified field.
504     * <p>
505     * The range object expresses the minimum and maximum valid values for a field.
506     * This instant is used to enhance the accuracy of the returned range.
507     * If it is not possible to return the range, because the field is not supported
508     * or for some other reason, an exception is thrown.
509     * <p>
510     * If the field is a {@link ChronoField} then the query is implemented here.
511     * The {@link #isSupported(TemporalField) supported fields} will return
512     * appropriate range instances.
513     * All other {@code ChronoField} instances will throw an {@code
514         UnsupportedOperationException}.
515     * <p>
516     * If the field is not a {@code ChronoField}, then the result of this method
517     * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
518     * passing {@code this} as the argument.
519     * Whether the range can be obtained is determined by the field.
520     *
521     * @param field the field to query the range for, not null
522     * @return the range of valid values for the field, not null
523     * @throws DateTimeException if the range for the field cannot be obtained
524     * @throws UnsupportedOperationException if the field is not supported
525     */
526     @Override // override for Javadoc
527     public ValueRange range(TemporalField field) {
528         return Temporal.super.range(field);
529     }
530
531     /**
532     * Gets the value of the specified field from this instant as an {@code int}.
533     * <p>
534     * This queries this instant for the value of the specified field.
535     * The returned value will always be within the valid range of values for the field.
536     * If it is not possible to return the value, because the field is not supported
537     * or for some other reason, an exception is thrown.

```

```

537 * <p>
538 * If the field is a {@link ChronoField} then the query is implemented here.
539 * The {@link #isSupported(TemporalField) supported fields} will return valid
540 * values based on this date-time, except {@code INSTANT_SECONDS} which is too
541 * large to fit in an {@code int} and throws a {@code DateTimeException}.
542 * All other {@code ChronoField} instances will throw an {@code
    UnsupportedTemporalTypeException}.
543 * <p>
544 * If the field is not a {@code ChronoField}, then the result of this method
545 * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
546 * passing {@code this} as the argument. Whether the value can be obtained,
547 * and what the value represents, is determined by the field.
548 *
549 * @param field the field to get, not null
550 * @return the value for the field
551 * @throws DateTimeException if a value for the field cannot be obtained or
552 *         the value is outside the range of valid values for the field
553 * @throws UnsupportedTemporalTypeException if the field is not supported or
554 *         the range of values exceeds an {@code int}
555 * @throws ArithmeticException if numeric overflow occurs
556 */
557 @Override // override for Javadoc and performance
558 public int get(TemporalField field) {
559     if (field instanceof ChronoField) {
560         switch ((ChronoField) field) {
561             case NANO_OF_SECOND: return nanos;
562             case MICRO_OF_SECOND: return nanos / 1000;
563             case MILLI_OF_SECOND: return nanos / 1000_000;
564             case INSTANT_SECONDS: INSTANT_SECONDS.checkValidIntValue(seconds);
565         }
566         throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
567     }
568     return range(field).checkValidIntValue(field.getFrom(this), field);
569 }
570
571 /**
572 * Gets the value of the specified field from this instant as a {@code long}.
573 * <p>
574 * This queries this instant for the value of the specified field.
575 * If it is not possible to return the value, because the field is not supported
576 * or for some other reason, an exception is thrown.
577 * <p>
578 * If the field is a {@link ChronoField} then the query is implemented here.
579 * The {@link #isSupported(TemporalField) supported fields} will return valid
580 * values based on this date-time.
581 * All other {@code ChronoField} instances will throw an {@code
    UnsupportedTemporalTypeException}.
582 * <p>
583 * If the field is not a {@code ChronoField}, then the result of this method
584 * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
585 * passing {@code this} as the argument. Whether the value can be obtained,
586 * and what the value represents, is determined by the field.
587 *

```



```

588     * @param field the field to get, not null
589     * @return the value for the field
590     * @throws DateTimeException if a value for the field cannot be obtained
591     * @throws UnsupportedTemporalTypeException if the field is not supported
592     * @throws ArithmeticException if numeric overflow occurs
593     */
594     @Override
595     public long getLong(TemporalField field) {
596         if (field instanceof ChronoField) {
597             switch ((ChronoField) field) {
598                 case NANO_OF_SECOND: return nanos;
599                 case MICRO_OF_SECOND: return nanos / 1000;
600                 case MILLI_OF_SECOND: return nanos / 1000_000;
601                 case INSTANT_SECONDS: return seconds;
602             }
603             throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
604         }
605         return field.getFrom(this);
606     }
607
608     //-----
609     /**
610     * Gets the number of seconds from the Java epoch of 1970-01-01T00:00:00Z.
611     * <p>
612     * The epoch second count is a simple incrementing count of seconds where
613     * second 0 is 1970-01-01T00:00:00Z.
614     * The nanosecond part of the day is returned by {@code getNanosOfSecond}.
615     *
616     * @return the seconds from the epoch of 1970-01-01T00:00:00Z
617     */
618     public long getEpochSecond() {
619         return seconds;
620     }
621
622     /**
623     * Gets the number of nanoseconds, later along the time-line, from the start
624     * of the second.
625     * <p>
626     * The nanosecond-of-second value measures the total number of nanoseconds from
627     * the second returned by {@code getEpochSecond}.
628     *
629     * @return the nanoseconds within the second, always positive, never exceeds 999,999,999
630     */
631     public int getNano() {
632         return nanos;
633     }
634
635     //-----
636     /**
637     * Returns an adjusted copy of this instant.
638     * <p>
639     * This returns an {@code Instant}, based on this one, with the instant adjusted.
640     * The adjustment takes place using the specified adjuster strategy object.

```

```

641     * Read the documentation of the adjuster to understand what adjustment will be made.
642     * <p>
643     * The result of this method is obtained by invoking the
644     * {@link TemporalAdjuster#adjustInto(Temporal)} method on the
645     * specified adjuster passing {@code this} as the argument.
646     * <p>
647     * This instance is immutable and unaffected by this method call.
648     *
649     * @param adjuster the adjuster to use, not null
650     * @return an {@code Instant} based on {@code this} with the adjustment made, not null
651     * @throws DateTimeException if the adjustment cannot be made
652     * @throws ArithmeticException if numeric overflow occurs
653     */
654     @Override
655     public Instant with(TemporalAdjuster adjuster) {
656         return (Instant) adjuster.adjustInto(this);
657     }
658
659     /**
660     * Returns a copy of this instant with the specified field set to a new value.
661     * <p>
662     * This returns an {@code Instant}, based on this one, with the value
663     * for the specified field changed.
664     * If it is not possible to set the value, because the field is not supported or for
665     * some other reason, an exception is thrown.
666     * <p>
667     * If the field is a {@link ChronoField} then the adjustment is implemented here.
668     * The supported fields behave as follows:
669     * <ul>
670     * <li>{@code NANO_OF_SECOND} -
671     * Returns an {@code Instant} with the specified nano-of-second.
672     * The epoch-second will be unchanged.
673     * <li>{@code MICRO_OF_SECOND} -
674     * Returns an {@code Instant} with the nano-of-second replaced by the specified
675     * micro-of-second multiplied by 1,000. The epoch-second will be unchanged.
676     * <li>{@code MILLI_OF_SECOND} -
677     * Returns an {@code Instant} with the nano-of-second replaced by the specified
678     * milli-of-second multiplied by 1,000,000. The epoch-second will be unchanged.
679     * <li>{@code INSTANT_SECONDS} -
680     * Returns an {@code Instant} with the specified epoch-second.
681     * The nano-of-second will be unchanged.
682     * </ul>
683     * <p>
684     * In all cases, if the new value is outside the valid range of values for the field
685     * then a {@code DateTimeException} will be thrown.
686     * <p>
687     * All other {@code ChronoField} instances will throw an {@code
688         UnsupportedTemporalTypeException}.
689     * <p>
690     * If the field is not a {@code ChronoField}, then the result of this method
691     * is obtained by invoking {@code TemporalField.adjustInto(Temporal, long)}
692     * passing {@code this} as the argument. In this case, the field determines

```

```

693     * <p>
694     * This instance is immutable and unaffected by this method call.
695     *
696     * @param field the field to set in the result, not null
697     * @param newValue the new value of the field in the result
698     * @return an {@code Instant} based on {@code this} with the specified field set, not null
699     * @throws DateTimeException if the field cannot be set
700     * @throws UnsupportedTemporalTypeException if the field is not supported
701     * @throws ArithmeticException if numeric overflow occurs
702     */
703     @Override
704     public Instant with(TemporalField field, long newValue) {
705         if (field instanceof ChronoField) {
706             ChronoField f = (ChronoField) field;
707             f.checkValidValue(newValue);
708             switch (f) {
709                 case MILLI_OF_SECOND: {
710                     int nval = (int) newValue * 1000_000;
711                     return (nval != nanos ? create(seconds, nval) : this);
712                 }
713                 case MICRO_OF_SECOND: {
714                     int nval = (int) newValue * 1000;
715                     return (nval != nanos ? create(seconds, nval) : this);
716                 }
717                 case NANO_OF_SECOND: return (newValue != nanos ? create(seconds, (int) newValue) :
718                     this);
719                 case INSTANT_SECONDS: return (newValue != seconds ? create(newValue, nanos) : this);
720             }
721             throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
722         }
723         return field.adjustInto(this, newValue);
724     }
725
726     //-----
727     /**
728     * Returns a copy of this {@code Instant} truncated to the specified unit.
729     * <p>
730     * Truncating the instant returns a copy of the original with fields
731     * smaller than the specified unit set to zero.
732     * The fields are calculated on the basis of using a UTC offset as seen
733     * in {@code toString}.
734     * For example, truncating with the {@link ChronoUnit#MINUTES MINUTES} unit will
735     * round down to the nearest minute, setting the seconds and nanoseconds to zero.
736     * <p>
737     * The unit must have a {@linkplain TemporalUnit#getDuration() duration}
738     * that divides into the length of a standard day without remainder.
739     * This includes all supplied time units on {@link ChronoUnit} and
740     * {@link ChronoUnit#DAYS DAYS}. Other units throw an exception.
741     * <p>
742     * This instance is immutable and unaffected by this method call.
743     *
744     * @param unit the unit to truncate to, not null

```

```

744     * @return an {@code Instant} based on this instant with the time truncated, not null
745     * @throws DateTimeException if the unit is invalid for truncation
746     * @throws UnsupportedTemporalTypeException if the unit is not supported
747     */
748     public Instant truncatedTo(TemporalUnit unit) {
749         if (unit == ChronoUnit.NANOS) {
750             return this;
751         }
752         Duration unitDur = unit.getDuration();
753         if (unitDur.getSeconds() > LocalTime.SECONDS_PER_DAY) {
754             throw new UnsupportedTemporalTypeException("Unit is too large to be used for truncation");
755         }
756         long dur = unitDur.toNanos();
757         if ((LocalTime.NANOS_PER_DAY % dur) != 0) {
758             throw new UnsupportedTemporalTypeException("Unit must divide into a standard day without remainder");
759         }
760         long nod = (seconds % LocalTime.SECONDS_PER_DAY) * LocalTime.NANOS_PER_SECOND + nanos;
761         long result = (nod / dur) * dur;
762         return plusNanos(result - nod);
763     }
764
765     //-----
766     /**
767     * Returns a copy of this instant with the specified amount added.
768     * <p>
769     * This returns an {@code Instant}, based on this one, with the specified amount added.
770     * The amount is typically {@link Duration} but may be any other type implementing
771     * the {@link TemporalAmount} interface.
772     * <p>
773     * The calculation is delegated to the amount object by calling
774     * {@link TemporalAmount#addTo(Temporal)}. The amount implementation is free
775     * to implement the addition in any way it wishes, however it typically
776     * calls back to {@link #plus(long, TemporalUnit)}. Consult the documentation
777     * of the amount implementation to determine if it can be successfully added.
778     * <p>
779     * This instance is immutable and unaffected by this method call.
780     *
781     * @param amountToAdd the amount to add, not null
782     * @return an {@code Instant} based on this instant with the addition made, not null
783     * @throws DateTimeException if the addition cannot be made
784     * @throws ArithmeticException if numeric overflow occurs
785     */
786     @Override
787     public Instant plus(TemporalAmount amountToAdd) {
788         return (Instant) amountToAdd.addTo(this);
789     }
790
791     /**
792     * Returns a copy of this instant with the specified amount added.
793     * <p>
794     * This returns an {@code Instant}, based on this one, with the amount

```

```

795 * in terms of the unit added. If it is not possible to add the amount, because the
796 * unit is not supported or for some other reason, an exception is thrown.
797 * <p>
798 * If the field is a {@link ChronoUnit} then the addition is implemented here.
799 * The supported fields behave as follows:
800 * <ul>
801 * <li>{@code NANOS} -
802 * Returns a {@code Instant} with the specified number of nanoseconds added.
803 * This is equivalent to {@link #plusNanos(long)}.
804 * <li>{@code MICROS} -
805 * Returns a {@code Instant} with the specified number of microseconds added.
806 * This is equivalent to {@link #plusNanos(long)} with the amount
807 * multiplied by 1,000.
808 * <li>{@code MILLIS} -
809 * Returns a {@code Instant} with the specified number of milliseconds added.
810 * This is equivalent to {@link #plusNanos(long)} with the amount
811 * multiplied by 1,000,000.
812 * <li>{@code SECONDS} -
813 * Returns a {@code Instant} with the specified number of seconds added.
814 * This is equivalent to {@link #plusSeconds(long)}.
815 * <li>{@code MINUTES} -
816 * Returns a {@code Instant} with the specified number of minutes added.
817 * This is equivalent to {@link #plusSeconds(long)} with the amount
818 * multiplied by 60.
819 * <li>{@code HOURS} -
820 * Returns a {@code Instant} with the specified number of hours added.
821 * This is equivalent to {@link #plusSeconds(long)} with the amount
822 * multiplied by 3,600.
823 * <li>{@code HALF_DAYS} -
824 * Returns a {@code Instant} with the specified number of half-days added.
825 * This is equivalent to {@link #plusSeconds(long)} with the amount
826 * multiplied by 43,200 (12 hours).
827 * <li>{@code DAYS} -
828 * Returns a {@code Instant} with the specified number of days added.
829 * This is equivalent to {@link #plusSeconds(long)} with the amount
830 * multiplied by 86,400 (24 hours).
831 * </ul>
832 * <p>
833 * All other {@code ChronoUnit} instances will throw an {@code UnsupportedOperationException}
834 *   }.
835 * <p>
836 * If the field is not a {@code ChronoUnit}, then the result of this method
837 * is obtained by invoking {@code TemporalUnit.addTo(Temporal, long)}
838 * passing {@code this} as the argument. In this case, the unit determines
839 * whether and how to perform the addition.
840 * <p>
841 * This instance is immutable and unaffected by this method call.
842 *
843 * @param amountToAdd the amount of the unit to add to the result, may be negative
844 * @param unit the unit of the amount to add, not null
845 * @return an {@code Instant} based on this instant with the specified amount added, not null
846 * @throws DateTimeException if the addition cannot be made
847 * @throws UnsupportedOperationException if the unit is not supported

```

```

847     * @throws ArithmeticException if numeric overflow occurs
848     */
849     @Override
850     public Instant plus(long amountToAdd, TemporalUnit unit) {
851         if (unit instanceof ChronoUnit) {
852             switch ((ChronoUnit) unit) {
853                 case NANOS: return plusNanos(amountToAdd);
854                 case MICROS: return plus(amountToAdd / 1000_000, (amountToAdd % 1000_000) * 1000);
855                 case MILLIS: return plusMillis(amountToAdd);
856                 case SECONDS: return plusSeconds(amountToAdd);
857                 case MINUTES: return plusSeconds(Math.multiplyExact(amountToAdd, SECONDS_PER_MINUTE));
858                 case HOURS: return plusSeconds(Math.multiplyExact(amountToAdd, SECONDS_PER_HOUR));
859                 case HALF_DAYS: return plusSeconds(Math.multiplyExact(amountToAdd, SECONDS_PER_DAY / 2));
860                 case DAYS: return plusSeconds(Math.multiplyExact(amountToAdd, SECONDS_PER_DAY));
861             }
862             throw new UnsupportedTemporalTypeException("Unsupported unit: " + unit);
863         }
864         return unit.addTo(this, amountToAdd);
865     }
866
867     //-----
868     /**
869     * Returns a copy of this instant with the specified duration in seconds added.
870     * <p>
871     * This instance is immutable and unaffected by this method call.
872     *
873     * @param secondsToAdd the seconds to add, positive or negative
874     * @return an {@code Instant} based on this instant with the specified seconds added, not null
875     * @throws DateTimeException if the result exceeds the maximum or minimum instant
876     * @throws ArithmeticException if numeric overflow occurs
877     */
878     public Instant plusSeconds(long secondsToAdd) {
879         return plus(secondsToAdd, 0);
880     }
881
882     /**
883     * Returns a copy of this instant with the specified duration in milliseconds added.
884     * <p>
885     * This instance is immutable and unaffected by this method call.
886     *
887     * @param millisToAdd the milliseconds to add, positive or negative
888     * @return an {@code Instant} based on this instant with the specified milliseconds added, not null
889     * @throws DateTimeException if the result exceeds the maximum or minimum instant
890     * @throws ArithmeticException if numeric overflow occurs
891     */
892     public Instant plusMillis(long millisToAdd) {
893         return plus(millisToAdd / 1000, (millisToAdd % 1000) * 1000_000);
894     }
895
896     /**

```

```

897     * Returns a copy of this instant with the specified duration in nanoseconds added.
898     * <p>
899     * This instance is immutable and unaffected by this method call.
900     *
901     * @param nanosToAdd the nanoseconds to add, positive or negative
902     * @return an {@code Instant} based on this instant with the specified nanoseconds added, not
903     *         null
904     * @throws DateTimeException if the result exceeds the maximum or minimum instant
905     * @throws ArithmeticException if numeric overflow occurs
906     */
907     public Instant plusNanos(long nanosToAdd) {
908         return plus(0, nanosToAdd);
909     }
910
911     /**
912     * Returns a copy of this instant with the specified duration added.
913     * <p>
914     * This instance is immutable and unaffected by this method call.
915     *
916     * @param secondsToAdd the seconds to add, positive or negative
917     * @param nanosToAdd the nanos to add, positive or negative
918     * @return an {@code Instant} based on this instant with the specified seconds added, not null
919     * @throws DateTimeException if the result exceeds the maximum or minimum instant
920     * @throws ArithmeticException if numeric overflow occurs
921     */
922     private Instant plus(long secondsToAdd, long nanosToAdd) {
923         if ((secondsToAdd | nanosToAdd) == 0) {
924             return this;
925         }
926         long epochSec = Math.addExact(seconds, secondsToAdd);
927         epochSec = Math.addExact(epochSec, nanosToAdd / NANOS_PER_SECOND);
928         nanosToAdd = nanosToAdd % NANOS_PER_SECOND;
929         long nanoAdjustment = nanos + nanosToAdd; // safe int+NANOS_PER_SECOND
930         return ofEpochSecond(epochSec, nanoAdjustment);
931     }
932
933     //-----
934     /**
935     * Returns a copy of this instant with the specified amount subtracted.
936     * <p>
937     * This returns an {@code Instant}, based on this one, with the specified amount subtracted.
938     * The amount is typically {@link Duration} but may be any other type implementing
939     * the {@link TemporalAmount} interface.
940     * <p>
941     * The calculation is delegated to the amount object by calling
942     * {@link TemporalAmount#subtractFrom(Temporal)}. The amount implementation is free
943     * to implement the subtraction in any way it wishes, however it typically
944     * calls back to {@link #minus(long, TemporalUnit)}. Consult the documentation
945     * of the amount implementation to determine if it can be successfully subtracted.
946     * <p>
947     * This instance is immutable and unaffected by this method call.
948     *
949     * @param amountToSubtract the amount to subtract, not null

```



```

949     * @return an {@code Instant} based on this instant with the subtraction made, not null
950     * @throws DateTimeException if the subtraction cannot be made
951     * @throws ArithmeticException if numeric overflow occurs
952     */
953     @Override
954     public Instant minus(TemporalAmount amountToSubtract) {
955         return (Instant) amountToSubtract.subtractFrom(this);
956     }
957
958     /**
959     * Returns a copy of this instant with the specified amount subtracted.
960     * <p>
961     * This returns a {@code Instant}, based on this one, with the amount
962     * in terms of the unit subtracted. If it is not possible to subtract the amount,
963     * because the unit is not supported or for some other reason, an exception is thrown.
964     * <p>
965     * This method is equivalent to {@link #plus(long, TemporalUnit)} with the amount negated.
966     * See that method for a full description of how addition, and thus subtraction, works.
967     * <p>
968     * This instance is immutable and unaffected by this method call.
969     *
970     * @param amountToSubtract the amount of the unit to subtract from the result, may be negative
971     * @param unit the unit of the amount to subtract, not null
972     * @return an {@code Instant} based on this instant with the specified amount subtracted, not
973     *         null
974     * @throws DateTimeException if the subtraction cannot be made
975     * @throws UnsupportedTemporalTypeException if the unit is not supported
976     * @throws ArithmeticException if numeric overflow occurs
977     */
978     @Override
979     public Instant minus(long amountToSubtract, TemporalUnit unit) {
980         return (amountToSubtract == Long.MIN_VALUE ? plus(Long.MAX_VALUE, unit).plus(1, unit) :
981             plus(-amountToSubtract, unit));
982     }
983
984     //-----
985     /**
986     * Returns a copy of this instant with the specified duration in seconds subtracted.
987     * <p>
988     * This instance is immutable and unaffected by this method call.
989     *
990     * @param secondsToSubtract the seconds to subtract, positive or negative
991     * @return an {@code Instant} based on this instant with the specified seconds subtracted, not
992     *         null
993     * @throws DateTimeException if the result exceeds the maximum or minimum instant
994     * @throws ArithmeticException if numeric overflow occurs
995     */
996     public Instant minusSeconds(long secondsToSubtract) {
997         if (secondsToSubtract == Long.MIN_VALUE) {
998             return plusSeconds(Long.MAX_VALUE).plusSeconds(1);
999         }
1000         return plusSeconds(-secondsToSubtract);
1001     }

```



```

999
1000 /**
1001  * Returns a copy of this instant with the specified duration in milliseconds subtracted.
1002  * <p>
1003  * This instance is immutable and unaffected by this method call.
1004  *
1005  * @param millisToSubtract the milliseconds to subtract, positive or negative
1006  * @return an {@code Instant} based on this instant with the specified milliseconds subtracted,
1007         not null
1008  * @throws DateTimeException if the result exceeds the maximum or minimum instant
1009  * @throws ArithmeticException if numeric overflow occurs
1010  */
1011 public Instant minusMillis(long millisToSubtract) {
1012     if (millisToSubtract == Long.MIN_VALUE) {
1013         return plusMillis(Long.MAX_VALUE).plusMillis(1);
1014     }
1015     return plusMillis(-millisToSubtract);
1016 }
1017
1018 /**
1019  * Returns a copy of this instant with the specified duration in nanoseconds subtracted.
1020  * <p>
1021  * This instance is immutable and unaffected by this method call.
1022  *
1023  * @param nanosToSubtract the nanoseconds to subtract, positive or negative
1024  * @return an {@code Instant} based on this instant with the specified nanoseconds subtracted,
1025         not null
1026  * @throws DateTimeException if the result exceeds the maximum or minimum instant
1027  * @throws ArithmeticException if numeric overflow occurs
1028  */
1029 public Instant minusNanos(long nanosToSubtract) {
1030     if (nanosToSubtract == Long.MIN_VALUE) {
1031         return plusNanos(Long.MAX_VALUE).plusNanos(1);
1032     }
1033     return plusNanos(-nanosToSubtract);
1034 }
1035
1036 //-----
1037 /**
1038  * Queries this instant using the specified query.
1039  * <p>
1040  * This queries this instant using the specified query strategy object.
1041  * The {@code TemporalQuery} object defines the logic to be used to
1042  * obtain the result. Read the documentation of the query to understand
1043  * what the result of this method will be.
1044  * <p>
1045  * The result of this method is obtained by invoking the
1046  * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
1047  * specified query passing {@code this} as the argument.
1048  *
1049  * @param <R> the type of the result
1050  * @param query the query to invoke, not null
1051  * @return the query result, null may be returned (defined by the query)

```

```

1050     * @throws DateTimeException if unable to query (defined by the query)
1051     * @throws ArithmeticException if numeric overflow occurs (defined by the query)
1052     */
1053     @SuppressWarnings("unchecked")
1054     @Override
1055     public <R> R query(TemporalQuery<R> query) {
1056         if (query == TemporalQueries.precision()) {
1057             return (R) NANOS;
1058         }
1059         // inline TemporalAccessor.super.query(query) as an optimization
1060         if (query == TemporalQueries.chronology() || query == TemporalQueries.zoneId() ||
1061             query == TemporalQueries.zone() || query == TemporalQueries.offset() ||
1062             query == TemporalQueries.localDate() || query == TemporalQueries.localTime()) {
1063             return null;
1064         }
1065         return query.queryFrom(this);
1066     }
1067
1068     /**
1069     * Adjusts the specified temporal object to have this instant.
1070     * <p>
1071     * This returns a temporal object of the same observable type as the input
1072     * with the instant changed to be the same as this.
1073     * <p>
1074     * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
1075     * twice, passing {@link ChronoField#INSTANT_SECONDS} and
1076     * {@link ChronoField#NANO_OF_SECOND} as the fields.
1077     * <p>
1078     * In most cases, it is clearer to reverse the calling pattern by using
1079     * {@link Temporal#with(TemporalAdjuster)}:
1080     * <pre>
1081     * // these two lines are equivalent, but the second approach is recommended
1082     * temporal = thisInstant.adjustInto(temporal);
1083     * temporal = temporal.with(thisInstant);
1084     * </pre>
1085     * <p>
1086     * This instance is immutable and unaffected by this method call.
1087     *
1088     * @param temporal the target object to be adjusted, not null
1089     * @return the adjusted object, not null
1090     * @throws DateTimeException if unable to make the adjustment
1091     * @throws ArithmeticException if numeric overflow occurs
1092     */
1093     @Override
1094     public Temporal adjustInto(Temporal temporal) {
1095         return temporal.with(INSTANT_SECONDS, seconds).with(NANO_OF_SECOND, nanos);
1096     }
1097
1098     /**
1099     * Calculates the amount of time until another instant in terms of the specified unit.
1100     * <p>
1101     * This calculates the amount of time between two {@code Instant}
1102     * objects in terms of a single {@code TemporalUnit}.

```

```

1103 * The start and end points are {@code this} and the specified instant.
1104 * The result will be negative if the end is before the start.
1105 * The calculation returns a whole number, representing the number of
1106 * complete units between the two instants.
1107 * The {@code Temporal} passed to this method is converted to a
1108 * {@code Instant} using {@link #from(TemporalAccessor)}.
1109 * For example, the amount in days between two dates can be calculated
1110 * using {@code startInstant.until(endInstant, SECONDS)}.
1111 * <p>
1112 * There are two equivalent ways of using this method.
1113 * The first is to invoke this method.
1114 * The second is to use {@link TemporalUnit#between(Temporal, Temporal)}:
1115 * <pre>
1116 *     // these two lines are equivalent
1117 *     amount = start.until(end, SECONDS);
1118 *     amount = SECONDS.between(start, end);
1119 * </pre>
1120 * The choice should be made based on which makes the code more readable.
1121 * <p>
1122 * The calculation is implemented in this method for {@link ChronoUnit}.
1123 * The units {@code NANOS}, {@code MICROS}, {@code MILLIS}, {@code SECONDS},
1124 * {@code MINUTES}, {@code HOURS}, {@code HALF_DAYS} and {@code DAYS}
1125 * are supported. Other {@code ChronoUnit} values will throw an exception.
1126 * <p>
1127 * If the unit is not a {@code ChronoUnit}, then the result of this method
1128 * is obtained by invoking {@code TemporalUnit.between(Temporal, Temporal)}
1129 * passing {@code this} as the first argument and the converted input temporal
1130 * as the second argument.
1131 * <p>
1132 * This instance is immutable and unaffected by this method call.
1133 *
1134 * @param endExclusive the end date, exclusive, which is converted to an {@code Instant}, not
1135 *     null
1136 * @param unit the unit to measure the amount in, not null
1137 * @return the amount of time between this instant and the end instant
1138 * @throws DateTimeException if the amount cannot be calculated, or the end
1139 *     temporal cannot be converted to an {@code Instant}
1140 * @throws UnsupportedTemporalTypeException if the unit is not supported
1141 * @throws ArithmeticException if numeric overflow occurs
1142 */
1143 @Override
1144 public long until(Temporal endExclusive, TemporalUnit unit) {
1145     Instant end = Instant.from(endExclusive);
1146     if (unit instanceof ChronoUnit) {
1147         ChronoUnit f = (ChronoUnit) unit;
1148         switch (f) {
1149             case NANOS: return nanosUntil(end);
1150             case MICROS: return nanosUntil(end) / 1000;
1151             case MILLIS: return Math.subtractExact(end.toEpochMilli(), toEpochMilli());
1152             case SECONDS: return secondsUntil(end);
1153             case MINUTES: return secondsUntil(end) / SECONDS_PER_MINUTE;
1154             case HOURS: return secondsUntil(end) / SECONDS_PER_HOUR;
1155             case HALF_DAYS: return secondsUntil(end) / (12 * SECONDS_PER_HOUR);

```

```

1155         case DAYS: return secondsUntil(end) / (SECONDS_PER_DAY);
1156     }
1157     throw new UnsupportedOperationException("Unsupported unit: " + unit);
1158 }
1159 return unit.between(this, end);
1160 }
1161
1162 private long nanosUntil(Instant end) {
1163     long secsDiff = Math.subtractExact(end.seconds, seconds);
1164     long totalNanos = Math.multiplyExact(secsDiff, NANOS_PER_SECOND);
1165     return Math.addExact(totalNanos, end.nanos - nanos);
1166 }
1167
1168 private long secondsUntil(Instant end) {
1169     long secsDiff = Math.subtractExact(end.seconds, seconds);
1170     long nanosDiff = end.nanos - nanos;
1171     if (secsDiff > 0 && nanosDiff < 0) {
1172         secsDiff--;
1173     } else if (secsDiff < 0 && nanosDiff > 0) {
1174         secsDiff++;
1175     }
1176     return secsDiff;
1177 }
1178
1179 //-----
1180 /**
1181  * Combines this instant with an offset to create an {@code OffsetDateTime}.
1182  * <p>
1183  * This returns an {@code OffsetDateTime} formed from this instant at the
1184  * specified offset from UTC/Greenwich. An exception will be thrown if the
1185  * instant is too large to fit into an offset date-time.
1186  * <p>
1187  * This method is equivalent to
1188  * {@link OffsetDateTime#ofInstant(Instant, ZoneId) OffsetDateTime.ofInstant(this, offset)}.
1189  *
1190  * @param offset the offset to combine with, not null
1191  * @return the offset date-time formed from this instant and the specified offset, not null
1192  * @throws DateTimeException if the result exceeds the supported range
1193  */
1194 public OffsetDateTime atOffset(ZoneOffset offset) {
1195     return OffsetDateTime.ofInstant(this, offset);
1196 }
1197
1198 /**
1199  * Combines this instant with a time-zone to create a {@code ZonedDateTime}.
1200  * <p>
1201  * This returns an {@code ZonedDateTime} formed from this instant at the
1202  * specified time-zone. An exception will be thrown if the instant is too
1203  * large to fit into a zoned date-time.
1204  * <p>
1205  * This method is equivalent to
1206  * {@link ZonedDateTime#ofInstant(Instant, ZoneId) ZonedDateTime.ofInstant(this, zone)}.
1207  *

```

```

1208     * @param zone the zone to combine with, not null
1209     * @return the zoned date-time formed from this instant and the specified zone, not null
1210     * @throws DateTimeException if the result exceeds the supported range
1211     */
1212     public ZonedDateTime atZone(ZoneId zone) {
1213         return ZonedDateTime.ofInstant(this, zone);
1214     }
1215
1216     //-----
1217     /**
1218     * Converts this instant to the number of milliseconds from the epoch
1219     * of 1970-01-01T00:00:00Z.
1220     * <p>
1221     * If this instant represents a point on the time-line too far in the future
1222     * or past to fit in a {@code long} milliseconds, then an exception is thrown.
1223     * <p>
1224     * If this instant has greater than millisecond precision, then the conversion
1225     * will drop any excess precision information as though the amount in nanoseconds
1226     * was subject to integer division by one million.
1227     *
1228     * @return the number of milliseconds since the epoch of 1970-01-01T00:00:00Z
1229     * @throws ArithmeticException if numeric overflow occurs
1230     */
1231     public long toEpochMilli() {
1232         if (seconds < 0 && nanos > 0) {
1233             long millis = Math.multiplyExact(seconds+1, 1000);
1234             long adjustment = nanos / 1000_000 - 1000;
1235             return Math.addExact(millis, adjustment);
1236         } else {
1237             long millis = Math.multiplyExact(seconds, 1000);
1238             return Math.addExact(millis, nanos / 1000_000);
1239         }
1240     }
1241
1242     //-----
1243     /**
1244     * Compares this instant to the specified instant.
1245     * <p>
1246     * The comparison is based on the time-line position of the instants.
1247     * It is "consistent with equals", as defined by {@link Comparable}.
1248     *
1249     * @param otherInstant the other instant to compare to, not null
1250     * @return the comparator value, negative if less, positive if greater
1251     * @throws NullPointerException if otherInstant is null
1252     */
1253     @Override
1254     public int compareTo(Instant otherInstant) {
1255         int cmp = Long.compare(seconds, otherInstant.seconds);
1256         if (cmp != 0) {
1257             return cmp;
1258         }
1259         return nanos - otherInstant.nanos;
1260     }

```

```

1261
1262 /**
1263  * Checks if this instant is after the specified instant.
1264  * <p>
1265  * The comparison is based on the time-line position of the instants.
1266  *
1267  * @param otherInstant the other instant to compare to, not null
1268  * @return true if this instant is after the specified instant
1269  * @throws NullPointerException if otherInstant is null
1270  */
1271 public boolean isAfter(Instant otherInstant) {
1272     return compareTo(otherInstant) > 0;
1273 }
1274
1275 /**
1276  * Checks if this instant is before the specified instant.
1277  * <p>
1278  * The comparison is based on the time-line position of the instants.
1279  *
1280  * @param otherInstant the other instant to compare to, not null
1281  * @return true if this instant is before the specified instant
1282  * @throws NullPointerException if otherInstant is null
1283  */
1284 public boolean isBefore(Instant otherInstant) {
1285     return compareTo(otherInstant) < 0;
1286 }
1287
1288 //-----
1289 /**
1290  * Checks if this instant is equal to the specified instant.
1291  * <p>
1292  * The comparison is based on the time-line position of the instants.
1293  *
1294  * @param otherInstant the other instant, null returns false
1295  * @return true if the other instant is equal to this one
1296  */
1297 @Override
1298 public boolean equals(Object otherInstant) {
1299     if (this == otherInstant) {
1300         return true;
1301     }
1302     if (otherInstant instanceof Instant) {
1303         Instant other = (Instant) otherInstant;
1304         return this.seconds == other.seconds &&
1305             this.nanos == other.nanos;
1306     }
1307     return false;
1308 }
1309
1310 /**
1311  * Returns a hash code for this instant.
1312  *
1313  * @return a suitable hash code

```

```

1314     */
1315     @Override
1316     public int hashCode() {
1317         return ((int) (seconds ^ (seconds >>> 32))) + 51 * nanos;
1318     }
1319
1320     //-----
1321     /**
1322      * A string representation of this instant using ISO-8601 representation.
1323      * <p>
1324      * The format used is the same as {@link DateTimeFormatter#ISO_INSTANT}.
1325      *
1326      * @return an ISO-8601 representation of this instant, not null
1327      */
1328     @Override
1329     public String toString() {
1330         return DateTimeFormatter.ISO_INSTANT.format(this);
1331     }
1332
1333     // -----
1334     /**
1335      * Writes the object using a
1336      * <a href="../../serialized-form.html#java.time.Ser">dedicated serialized form</a>.
1337      * @serialData
1338      * <pre>
1339      *   out.writeByte(2); // identifies an Instant
1340      *   out.writeLong(seconds);
1341      *   out.writeInt(nanos);
1342      * </pre>
1343      *
1344      * @return the instance of {@code Ser}, not null
1345      */
1346     private Object writeReplace() {
1347         return new Ser(Ser.INSTANT_TYPE, this);
1348     }
1349
1350     /**
1351      * Defend against malicious streams.
1352      *
1353      * @param s the stream to read
1354      * @throws InvalidObjectException always
1355      */
1356     private void readObject(ObjectInputStream s) throws InvalidObjectException {
1357         throw new InvalidObjectException("Deserialization via serialization delegate");
1358     }
1359
1360     void writeExternal(DataOutput out) throws IOException {
1361         out.writeLong(seconds);
1362         out.writeInt(nanos);
1363     }
1364
1365     static Instant readExternal(DataInput in) throws IOException {
1366         long seconds = in.readLong();

```

```
1367     int nanos = in.readInt();
1368     return Instant.ofEpochSecond(seconds, nanos);
1369 }
1370
1371 }
```

4.6 LocalDate.java

Secuencia 55: java/time/LocalDate.java

```
1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
```



```
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time;
63
64  import static java.time.LocalTime.SECONDS_PER_DAY;
65  import static java.time.temporal.ChronoField.ALIGNED_DAY_OF_WEEK_IN_MONTH;
66  import static java.time.temporal.ChronoField.ALIGNED_DAY_OF_WEEK_IN_YEAR;
67  import static java.time.temporal.ChronoField.ALIGNED_WEEK_OF_MONTH;
68  import static java.time.temporal.ChronoField.ALIGNED_WEEK_OF_YEAR;
69  import static java.time.temporal.ChronoField.DAY_OF_MONTH;
70  import static java.time.temporal.ChronoField.DAY_OF_YEAR;
71  import static java.time.temporal.ChronoField.EPOCH_DAY;
72  import static java.time.temporal.ChronoField.ERA;
73  import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
74  import static java.time.temporal.ChronoField.PROLEPTIC_MONTH;
75  import static java.time.temporal.ChronoField.YEAR;
76
77  import java.io.DataInput;
78  import java.io.DataOutput;
79  import java.io.IOException;
80  import java.io.InvalidObjectException;
81  import java.io.ObjectInputStream;
82  import java.io.Serializable;
83  import java.time.chrono.ChronoLocalDate;
84  import java.time.chrono.Era;
85  import java.time.chrono.IsoChronology;
86  import java.time.format.DateTimeFormatter;
87  import java.time.format.DateTimeParseException;
88  import java.time.temporal.ChronoField;
89  import java.time.temporal.ChronoUnit;
90  import java.time.temporal.Temporal;
91  import java.time.temporal.TemporalAccessor;
92  import java.time.temporal.TemporalAdjuster;
93  import java.time.temporal.TemporalAmount;
94  import java.time.temporal.TemporalField;
95  import java.time.temporal.TemporalQueries;
96  import java.time.temporal.TemporalQuery;
97  import java.time.temporal.TemporalUnit;
```

```

98 import java.time.temporal.UnsupportedTemporalTypeException;
99 import java.time.temporal.ValueRange;
100 import java.time.zone.ZoneOffsetTransition;
101 import java.time.zone.ZoneRules;
102 import java.util.Objects;
103
104 /**
105  * A date without a time-zone in the ISO-8601 calendar system,
106  * such as {@code 2007-12-03}.
107  * <p>
108  * {@code LocalDate} is an immutable date-time object that represents a date,
109  * often viewed as year-month-day. Other date fields, such as day-of-year,
110  * day-of-week and week-of-year, can also be accessed.
111  * For example, the value "2nd October 2007" can be stored in a {@code LocalDate}.
112  * <p>
113  * This class does not store or represent a time or time-zone.
114  * Instead, it is a description of the date, as used for birthdays.
115  * It cannot represent an instant on the time-line without additional information
116  * such as an offset or time-zone.
117  * <p>
118  * The ISO-8601 calendar system is the modern civil calendar system used today
119  * in most of the world. It is equivalent to the proleptic Gregorian calendar
120  * system, in which today's rules for leap years are applied for all time.
121  * For most applications written today, the ISO-8601 rules are entirely suitable.
122  * However, any application that makes use of historical dates, and requires them
123  * to be accurate will find the ISO-8601 approach unsuitable.
124  *
125  * <p>
126  * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
127  * class; use of identity-sensitive operations (including reference equality
128  * ({@code ==}), identity hash code, or synchronization) on instances of
129  * {@code LocalDate} may have unpredictable results and should be avoided.
130  * The {@code equals} method should be used for comparisons.
131  *
132  * @implSpec
133  * This class is immutable and thread-safe.
134  *
135  * @since 1.8
136  */
137 public final class LocalDate
138     implements Temporal, TemporalAdjuster, ChronoLocalDate, Serializable {
139
140     /**
141      * The minimum supported {@code LocalDate}, '-999999999-01-01'.
142      * This could be used by an application as a "far past" date.
143      */
144     public static final LocalDate MIN = LocalDate.of(Year.MIN_VALUE, 1, 1);
145
146     /**
147      * The maximum supported {@code LocalDate}, '+999999999-12-31'.
148      * This could be used by an application as a "far future" date.
149      */
150     public static final LocalDate MAX = LocalDate.of(Year.MAX_VALUE, 12, 31);

```

```

151  /**
152   * Serialization version.
153   */
154  private static final long serialVersionUID = 2942565459149668126L;
155  /**
156   * The number of days in a 400 year cycle.
157   */
158  private static final int DAYS_PER_CYCLE = 146097;
159  /**
160   * The number of days from year zero to year 1970.
161   * There are five 400 year cycles from year zero to 2000.
162   * There are 7 leap years from 1970 to 2000.
163   */
164  static final long DAYS_0000_TO_1970 = (DAYS_PER_CYCLE * 5L) - (30L * 365L + 7L);
165
166  /**
167   * The year.
168   */
169  private final int year;
170  /**
171   * The month-of-year.
172   */
173  private final short month;
174  /**
175   * The day-of-month.
176   */
177  private final short day;
178
179  //-----
180  /**
181   * Obtains the current date from the system clock in the default time-zone.
182   * <p>
183   * This will query the {@link Clock#systemDefaultZone() system clock} in the default
184   * time-zone to obtain the current date.
185   * <p>
186   * Using this method will prevent the ability to use an alternate clock for testing
187   * because the clock is hard-coded.
188   *
189   * @return the current date using the system clock and default time-zone, not null
190   */
191  public static LocalDate now() {
192      return now(Clock.systemDefaultZone());
193  }
194
195  /**
196   * Obtains the current date from the system clock in the specified time-zone.
197   * <p>
198   * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current date.
199   * Specifying the time-zone avoids dependence on the default time-zone.
200   * <p>
201   * Using this method will prevent the ability to use an alternate clock for testing
202   * because the clock is hard-coded.
203   *

```

```

204     * @param zone the zone ID to use, not null
205     * @return the current date using the system clock, not null
206     */
207     public static LocalDate now(ZoneId zone) {
208         return now(Clock.system(zone));
209     }
210
211     /**
212     * Obtains the current date from the specified clock.
213     * <p>
214     * This will query the specified clock to obtain the current date - today.
215     * Using this method allows the use of an alternate clock for testing.
216     * The alternate clock may be introduced using {@link Clock dependency injection}.
217     *
218     * @param clock the clock to use, not null
219     * @return the current date, not null
220     */
221     public static LocalDate now(Clock clock) {
222         Objects.requireNonNull(clock, "clock");
223         // inline to avoid creating object and Instant checks
224         final Instant now = clock.instant(); // called once
225         ZoneOffset offset = clock.getZone().getRules().getOffset(now);
226         long epochSec = now.getEpochSecond() + offset.getTotalSeconds(); // overflow caught later
227         long epochDay = Math.floorDiv(epochSec, SECONDS_PER_DAY);
228         return LocalDate.ofEpochDay(epochDay);
229     }
230
231     //-----
232     /**
233     * Obtains an instance of {@code LocalDate} from a year, month and day.
234     * <p>
235     * This returns a {@code LocalDate} with the specified year, month and day-of-month.
236     * The day must be valid for the year and month, otherwise an exception will be thrown.
237     *
238     * @param year the year to represent, from MIN_YEAR to MAX_YEAR
239     * @param month the month-of-year to represent, not null
240     * @param dayOfMonth the day-of-month to represent, from 1 to 31
241     * @return the local date, not null
242     * @throws DateTimeException if the value of any field is out of range,
243     * or if the day-of-month is invalid for the month-year
244     */
245     public static LocalDate of(int year, Month month, int dayOfMonth) {
246         YEAR.checkValidValue(year);
247         Objects.requireNonNull(month, "month");
248         DAY_OF_MONTH.checkValidValue(dayOfMonth);
249         return create(year, month.getValue(), dayOfMonth);
250     }
251
252     /**
253     * Obtains an instance of {@code LocalDate} from a year, month and day.
254     * <p>
255     * This returns a {@code LocalDate} with the specified year, month and day-of-month.
256     * The day must be valid for the year and month, otherwise an exception will be thrown.

```

```

257 *
258 * @param year the year to represent, from MIN_YEAR to MAX_YEAR
259 * @param month the month-of-year to represent, from 1 (January) to 12 (December)
260 * @param dayOfMonth the day-of-month to represent, from 1 to 31
261 * @return the local date, not null
262 * @throws DateTimeException if the value of any field is out of range,
263 * or if the day-of-month is invalid for the month-year
264 */
265 public static LocalDate of(int year, int month, int dayOfMonth) {
266     YEAR.checkValidValue(year);
267     MONTH_OF_YEAR.checkValidValue(month);
268     DAY_OF_MONTH.checkValidValue(dayOfMonth);
269     return create(year, month, dayOfMonth);
270 }
271
272 //-----
273 /**
274  * Obtains an instance of {@code LocalDate} from a year and day-of-year.
275  * <p>
276  * This returns a {@code LocalDate} with the specified year and day-of-year.
277  * The day-of-year must be valid for the year, otherwise an exception will be thrown.
278  *
279  * @param year the year to represent, from MIN_YEAR to MAX_YEAR
280  * @param dayOfYear the day-of-year to represent, from 1 to 366
281  * @return the local date, not null
282  * @throws DateTimeException if the value of any field is out of range,
283  * or if the day-of-year is invalid for the year
284  */
285 public static LocalDate ofYearDay(int year, int dayOfYear) {
286     YEAR.checkValidValue(year);
287     DAY_OF_YEAR.checkValidValue(dayOfYear);
288     boolean leap = IsoChronology.INSTANCE.isLeapYear(year);
289     if (dayOfYear == 366 && leap == false) {
290         throw new DateTimeException("Invalid date 'DayOfYear 366' as '" + year + "' is not a
291             leap year");
292     }
293     Month moy = Month.of((dayOfYear - 1) / 31 + 1);
294     int monthEnd = moy.firstDayOfYear(leap) + moy.length(leap) - 1;
295     if (dayOfYear > monthEnd) {
296         moy = moy.plus(1);
297     }
298     int dom = dayOfYear - moy.firstDayOfYear(leap) + 1;
299     return new LocalDate(year, moy.getValue(), dom);
300 }
301
302 //-----
303 /**
304  * Obtains an instance of {@code LocalDate} from the epoch day count.
305  * <p>
306  * This returns a {@code LocalDate} with the specified epoch-day.
307  * The {@link ChronoField#EPOCH_DAY EPOCH_DAY} is a simple incrementing count
308  * of days where day 0 is 1970-01-01. Negative numbers represent earlier days.
309  *

```

```

309     * @param epochDay the Epoch Day to convert, based on the epoch 1970-01-01
310     * @return the local date, not null
311     * @throws DateTimeException if the epoch day exceeds the supported date range
312     */
313     public static LocalDate ofEpochDay(long epochDay) {
314         long zeroDay = epochDay + DAYS_0000_TO_1970;
315         // find the march-based year
316         zeroDay -= 60; // adjust to 0000-03-01 so leap day is at end of four year cycle
317         long adjust = 0;
318         if (zeroDay < 0) {
319             // adjust negative years to positive for calculation
320             long adjustCycles = (zeroDay + 1) / DAYS_PER_CYCLE - 1;
321             adjust = adjustCycles * 400;
322             zeroDay += -adjustCycles * DAYS_PER_CYCLE;
323         }
324         long yearEst = (400 * zeroDay + 591) / DAYS_PER_CYCLE;
325         long doyEst = zeroDay - (365 * yearEst + yearEst / 4 - yearEst / 100 + yearEst / 400);
326         if (doyEst < 0) {
327             // fix estimate
328             yearEst--;
329             doyEst = zeroDay - (365 * yearEst + yearEst / 4 - yearEst / 100 + yearEst / 400);
330         }
331         yearEst += adjust; // reset any negative year
332         int marchDoy0 = (int) doyEst;
333
334         // convert march-based values back to january-based
335         int marchMonth0 = (marchDoy0 * 5 + 2) / 153;
336         int month = (marchMonth0 + 2) % 12 + 1;
337         int dom = marchDoy0 - (marchMonth0 * 306 + 5) / 10 + 1;
338         yearEst += marchMonth0 / 10;
339
340         // check year now we are certain it is correct
341         int year = YEAR.checkValidIntValue(yearEst);
342         return new LocalDate(year, month, dom);
343     }
344
345     //-----
346     /**
347     * Obtains an instance of {@code LocalDate} from a temporal object.
348     * <p>
349     * This obtains a local date based on the specified temporal.
350     * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
351     * which this factory converts to an instance of {@code LocalDate}.
352     * <p>
353     * The conversion uses the {@link TemporalQueries#localDate()} query, which relies
354     * on extracting the {@link ChronoField#EPOCH_DAY EPOCH_DAY} field.
355     * <p>
356     * This method matches the signature of the functional interface {@link TemporalQuery}
357     * allowing it to be used as a query via method reference, {@code LocalDate::from}.
358     *
359     * @param temporal the temporal object to convert, not null
360     * @return the local date, not null
361     * @throws DateTimeException if unable to convert to a {@code LocalDate}

```

```

362     */
363     public static LocalDate from(TemporalAccessor temporal) {
364         Objects.requireNonNull(temporal, "temporal");
365         LocalDate date = temporal.query(TemporalQueries.localDate());
366         if (date == null) {
367             throw new DateTimeException("Unable to obtain LocalDate from TemporalAccessor: " +
368                 temporal + " of type " + temporal.getClass().getName());
369         }
370         return date;
371     }
372
373     //-----
374     /**
375      * Obtains an instance of {@code LocalDate} from a text string such as {@code 2007-12-03}.
376      * <p>
377      * The string must represent a valid date and is parsed using
378      * {@link java.time.format.DateTimeFormatter#ISO_LOCAL_DATE}.
379      *
380      * @param text the text to parse such as "2007-12-03", not null
381      * @return the parsed local date, not null
382      * @throws DateTimeParseException if the text cannot be parsed
383      */
384     public static LocalDate parse(CharSequence text) {
385         return parse(text, DateTimeFormatter.ISO_LOCAL_DATE);
386     }
387
388     /**
389      * Obtains an instance of {@code LocalDate} from a text string using a specific formatter.
390      * <p>
391      * The text is parsed using the formatter, returning a date.
392      *
393      * @param text the text to parse, not null
394      * @param formatter the formatter to use, not null
395      * @return the parsed local date, not null
396      * @throws DateTimeParseException if the text cannot be parsed
397      */
398     public static LocalDate parse(CharSequence text, DateTimeFormatter formatter) {
399         Objects.requireNonNull(formatter, "formatter");
400         return formatter.parse(text, LocalDate::from);
401     }
402
403     //-----
404     /**
405      * Creates a local date from the year, month and day fields.
406      *
407      * @param year the year to represent, validated from MIN_YEAR to MAX_YEAR
408      * @param month the month-of-year to represent, from 1 to 12, validated
409      * @param dayOfMonth the day-of-month to represent, validated from 1 to 31
410      * @return the local date, not null
411      * @throws DateTimeException if the day-of-month is invalid for the month-year
412      */
413     private static LocalDate create(int year, int month, int dayOfMonth) {
414         if (dayOfMonth > 28) {

```

```

415         int dom = 31;
416         switch (month) {
417             case 2:
418                 dom = (IsoChronology.INSTANCE.isLeapYear(year) ? 29 : 28);
419                 break;
420             case 4:
421             case 6:
422             case 9:
423             case 11:
424                 dom = 30;
425                 break;
426         }
427         if (dayOfMonth > dom) {
428             if (dayOfMonth == 29) {
429                 throw new DateTimeException("Invalid date 'February 29' as '" + year + "' is
                                     not a leap year");
430             } else {
431                 throw new DateTimeException("Invalid date '" + Month.of(month).name() + " " +
                                     dayOfMonth + "'");
432             }
433         }
434     }
435     return new LocalDate(year, month, dayOfMonth);
436 }
437
438 /**
439  * Resolves the date, resolving days past the end of month.
440  *
441  * @param year the year to represent, validated from MIN_YEAR to MAX_YEAR
442  * @param month the month-of-year to represent, validated from 1 to 12
443  * @param day the day-of-month to represent, validated from 1 to 31
444  * @return the resolved date, not null
445  */
446 private static LocalDate resolvePreviousValid(int year, int month, int day) {
447     switch (month) {
448         case 2:
449             day = Math.min(day, IsoChronology.INSTANCE.isLeapYear(year) ? 29 : 28);
450             break;
451         case 4:
452         case 6:
453         case 9:
454         case 11:
455             day = Math.min(day, 30);
456             break;
457     }
458     return new LocalDate(year, month, day);
459 }
460
461 /**
462  * Constructor, previously validated.
463  *
464  * @param year the year to represent, from MIN_YEAR to MAX_YEAR
465  * @param month the month-of-year to represent, not null

```



```

466     * @param dayOfMonth the day-of-month to represent, valid for year-month, from 1 to 31
467     */
468     private LocalDate(int year, int month, int dayOfMonth) {
469         this.year = year;
470         this.month = (short) month;
471         this.day = (short) dayOfMonth;
472     }
473
474     //-----
475     /**
476      * Checks if the specified field is supported.
477      * <p>
478      * This checks if this date can be queried for the specified field.
479      * If false, then calling the {@link #range(TemporalField) range},
480      * {@link #get(TemporalField) get} and {@link #with(TemporalField, long)}
481      * methods will throw an exception.
482      * <p>
483      * If the field is a {@link ChronoField} then the query is implemented here.
484      * The supported fields are:
485      * <ul>
486      * <li>{@code DAY_OF_WEEK}
487      * <li>{@code ALIGNED_DAY_OF_WEEK_IN_MONTH}
488      * <li>{@code ALIGNED_DAY_OF_WEEK_IN_YEAR}
489      * <li>{@code DAY_OF_MONTH}
490      * <li>{@code DAY_OF_YEAR}
491      * <li>{@code EPOCH_DAY}
492      * <li>{@code ALIGNED_WEEK_OF_MONTH}
493      * <li>{@code ALIGNED_WEEK_OF_YEAR}
494      * <li>{@code MONTH_OF_YEAR}
495      * <li>{@code PROLEPTIC_MONTH}
496      * <li>{@code YEAR_OF_ERA}
497      * <li>{@code YEAR}
498      * <li>{@code ERA}
499      * </ul>
500      * All other {@code ChronoField} instances will return false.
501      * <p>
502      * If the field is not a {@code ChronoField}, then the result of this method
503      * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
504      * passing {@code this} as the argument.
505      * Whether the field is supported is determined by the field.
506      *
507      * @param field the field to check, null returns false
508      * @return true if the field is supported on this date, false if not
509      */
510     @Override // override for Javadoc
511     public boolean isSupported(TemporalField field) {
512         return ChronoLocalDate.super.isSupported(field);
513     }
514
515     /**
516      * Checks if the specified unit is supported.
517      * <p>
518      * This checks if the specified unit can be added to, or subtracted from, this date.

```

```

519 * If false, then calling the {@link #plus(long, TemporalUnit)} and
520 * {@link #minus(long, TemporalUnit) minus} methods will throw an exception.
521 * <p>
522 * If the unit is a {@link ChronoUnit} then the query is implemented here.
523 * The supported units are:
524 * <ul>
525 * <li>{@code DAYS}
526 * <li>{@code WEEKS}
527 * <li>{@code MONTHS}
528 * <li>{@code YEARS}
529 * <li>{@code DECADES}
530 * <li>{@code CENTURIES}
531 * <li>{@code MILLENNIA}
532 * <li>{@code ERAS}
533 * </ul>
534 * All other {@code ChronoUnit} instances will return false.
535 * <p>
536 * If the unit is not a {@code ChronoUnit}, then the result of this method
537 * is obtained by invoking {@code TemporalUnit.isSupportedBy(Temporal)}
538 * passing {@code this} as the argument.
539 * Whether the unit is supported is determined by the unit.
540 *
541 * @param unit the unit to check, null returns false
542 * @return true if the unit can be added/subtracted, false if not
543 */
544 @Override // override for Javadoc
545 public boolean isSupported(TemporalUnit unit) {
546     return ChronoLocalDate.super.isSupported(unit);
547 }
548
549 //-----
550 /**
551 * Gets the range of valid values for the specified field.
552 * <p>
553 * The range object expresses the minimum and maximum valid values for a field.
554 * This date is used to enhance the accuracy of the returned range.
555 * If it is not possible to return the range, because the field is not supported
556 * or for some other reason, an exception is thrown.
557 * <p>
558 * If the field is a {@link ChronoField} then the query is implemented here.
559 * The {@link #isSupported(TemporalField) supported fields} will return
560 * appropriate range instances.
561 * All other {@code ChronoField} instances will throw an {@code
562     UnsupportedTemporalTypeException}.
563 * <p>
564 * If the field is not a {@code ChronoField}, then the result of this method
565 * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
566 * passing {@code this} as the argument.
567 * Whether the range can be obtained is determined by the field.
568 *
569 * @param field the field to query the range for, not null
570 * @return the range of valid values for the field, not null
571 * @throws DateTimeException if the range for the field cannot be obtained

```

```

571     * @throws UnsupportedOperationException if the field is not supported
572     */
573     @Override
574     public ValueRange range(TemporalField field) {
575         if (field instanceof ChronoField) {
576             ChronoField f = (ChronoField) field;
577             if (f.isDateBased()) {
578                 switch (f) {
579                     case DAY_OF_MONTH: return ValueRange.of(1, lengthOfMonth());
580                     case DAY_OF_YEAR: return ValueRange.of(1, lengthOfYear());
581                     case ALIGNED_WEEK_OF_MONTH: return ValueRange.of(1, getMonth() == Month.
582                         FEBRUARY && isLeapYear() == false ? 4 : 5);
583                     case YEAR_OF_ERA:
584                         return (getYear() <= 0 ? ValueRange.of(1, Year.MAX_VALUE + 1) : ValueRange.
585                             of(1, Year.MAX_VALUE));
586                 }
587             }
588             return field.range();
589         }
590         throw new UnsupportedOperationException("Unsupported field: " + field);
591     }
592     /**
593     * Gets the value of the specified field from this date as an {@code int}.
594     * <p>
595     * This queries this date for the value of the specified field.
596     * The returned value will always be within the valid range of values for the field.
597     * If it is not possible to return the value, because the field is not supported
598     * or for some other reason, an exception is thrown.
599     * <p>
600     * If the field is a {@link ChronoField} then the query is implemented here.
601     * The {@link #isSupported(TemporalField) supported fields} will return valid
602     * values based on this date, except {@code EPOCH_DAY} and {@code PROLEPTIC_MONTH}
603     * which are too large to fit in an {@code int} and throw a {@code DateTimeException}.
604     * All other {@code ChronoField} instances will throw an {@code
605         UnsupportedOperationException}.
606     * <p>
607     * If the field is not a {@code ChronoField}, then the result of this method
608     * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
609     * passing {@code this} as the argument. Whether the value can be obtained,
610     * and what the value represents, is determined by the field.
611     *
612     * @param field the field to get, not null
613     * @return the value for the field
614     * @throws DateTimeException if a value for the field cannot be obtained or
615     *         the value is outside the range of valid values for the field
616     * @throws UnsupportedOperationException if the field is not supported or
617     *         the range of values exceeds an {@code int}
618     * @throws ArithmeticException if numeric overflow occurs
619     */
620     @Override // override for Javadoc and performance
621     public int get(TemporalField field) {

```

```

621     if (field instanceof ChronoField) {
622         return get0(field);
623     }
624     return ChronoLocalDate.super.get(field);
625 }
626
627 /**
628  * Gets the value of the specified field from this date as a {@code long}.
629  * <p>
630  * This queries this date for the value of the specified field.
631  * If it is not possible to return the value, because the field is not supported
632  * or for some other reason, an exception is thrown.
633  * <p>
634  * If the field is a {@link ChronoField} then the query is implemented here.
635  * The {@link #isSupported(TemporalField) supported fields} will return valid
636  * values based on this date.
637  * All other {@code ChronoField} instances will throw an {@code
        UnsupportedTemporalTypeException}.
638  * <p>
639  * If the field is not a {@code ChronoField}, then the result of this method
640  * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
641  * passing {@code this} as the argument. Whether the value can be obtained,
642  * and what the value represents, is determined by the field.
643  *
644  * @param field the field to get, not null
645  * @return the value for the field
646  * @throws DateTimeException if a value for the field cannot be obtained
647  * @throws UnsupportedTemporalTypeException if the field is not supported
648  * @throws ArithmeticException if numeric overflow occurs
649  */
650 @Override
651 public long getLong(TemporalField field) {
652     if (field instanceof ChronoField) {
653         if (field == EPOCH_DAY) {
654             return toEpochDay();
655         }
656         if (field == PROLEPTIC_MONTH) {
657             return getProlepticMonth();
658         }
659         return get0(field);
660     }
661     return field.getFrom(this);
662 }
663
664 private int get0(TemporalField field) {
665     switch ((ChronoField) field) {
666         case DAY_OF_WEEK: return getDayOfWeek().getValue();
667         case ALIGNED_DAY_OF_WEEK_IN_MONTH: return ((day - 1) % 7) + 1;
668         case ALIGNED_DAY_OF_WEEK_IN_YEAR: return ((getDayOfYear() - 1) % 7) + 1;
669         case DAY_OF_MONTH: return day;
670         case DAY_OF_YEAR: return getDayOfYear();
671         case EPOCH_DAY: throw new UnsupportedTemporalTypeException("Invalid field 'EpochDay'
            for get() method, use getLong() instead");
    }

```

```

672         case ALIGNED_WEEK_OF_MONTH: return ((day - 1) / 7) + 1;
673         case ALIGNED_WEEK_OF_YEAR: return ((getDayOfYear() - 1) / 7) + 1;
674         case MONTH_OF_YEAR: return month;
675         case PROLEPTIC_MONTH: throw new UnsupportedOperationException("Invalid field '
        ProlepticMonth' for get() method, use getLong() instead");
676         case YEAR_OF_ERA: return (year >= 1 ? year : 1 - year);
677         case YEAR: return year;
678         case ERA: return (year >= 1 ? 1 : 0);
679     }
680     throw new UnsupportedOperationException("Unsupported field: " + field);
681 }
682
683 private long getProlepticMonth() {
684     return (year * 12L + month - 1);
685 }
686
687 //-----
688 /**
689  * Gets the chronology of this date, which is the ISO calendar system.
690  * <p>
691  * The {@code Chronology} represents the calendar system in use.
692  * The ISO-8601 calendar system is the modern civil calendar system used today
693  * in most of the world. It is equivalent to the proleptic Gregorian calendar
694  * system, in which today's rules for leap years are applied for all time.
695  *
696  * @return the ISO chronology, not null
697  */
698 @Override
699 public IsoChronology getChronology() {
700     return IsoChronology.INSTANCE;
701 }
702
703 /**
704  * Gets the era applicable at this date.
705  * <p>
706  * The official ISO-8601 standard does not define eras, however {@code IsoChronology} does.
707  * It defines two eras, 'CE' from year one onwards and 'BCE' from year zero backwards.
708  * Since dates before the Julian-Gregorian cutover are not in line with history,
709  * the cutover between 'BCE' and 'CE' is also not aligned with the commonly used
710  * eras, often referred to using 'BC' and 'AD'.
711  * <p>
712  * Users of this class should typically ignore this method as it exists primarily
713  * to fulfill the {@link ChronoLocalDate} contract where it is necessary to support
714  * the Japanese calendar system.
715  * <p>
716  * The returned era will be a singleton capable of being compared with the constants
717  * in {@link IsoChronology} using the {@code ==} operator.
718  *
719  * @return the {@code IsoChronology} era constant applicable at this date, not null
720  */
721 @Override // override for Javadoc
722 public Era getEra() {
723     return ChronoLocalDate.super.getEra();

```

```
724     }
725
726     /**
727      * Gets the year field.
728      * <p>
729      * This method returns the primitive {@code int} value for the year.
730      * <p>
731      * The year returned by this method is proleptic as per {@code get(YEAR)}.
732      * To obtain the year-of-era, use {@code get(YEAR_OF_ERA)}.
733      *
734      * @return the year, from MIN_YEAR to MAX_YEAR
735      */
736     public int getYear() {
737         return year;
738     }
739
740     /**
741      * Gets the month-of-year field from 1 to 12.
742      * <p>
743      * This method returns the month as an {@code int} from 1 to 12.
744      * Application code is frequently clearer if the enum {@link Month}
745      * is used by calling {@link #getMonth()}.
746      *
747      * @return the month-of-year, from 1 to 12
748      * @see #getMonth()
749      */
750     public int getMonthValue() {
751         return month;
752     }
753
754     /**
755      * Gets the month-of-year field using the {@code Month} enum.
756      * <p>
757      * This method returns the enum {@link Month} for the month.
758      * This avoids confusion as to what {@code int} values mean.
759      * If you need access to the primitive {@code int} value then the enum
760      * provides the {@link Month#getValue() int value}.
761      *
762      * @return the month-of-year, not null
763      * @see #getMonthValue()
764      */
765     public Month getMonth() {
766         return Month.of(month);
767     }
768
769     /**
770      * Gets the day-of-month field.
771      * <p>
772      * This method returns the primitive {@code int} value for the day-of-month.
773      *
774      * @return the day-of-month, from 1 to 31
775      */
776     public int getDayOfMonth() {
```

```

777         return day;
778     }
779
780     /**
781      * Gets the day-of-year field.
782      * <p>
783      * This method returns the primitive {@code int} value for the day-of-year.
784      *
785      * @return the day-of-year, from 1 to 365, or 366 in a leap year
786      */
787     public int getDayOfYear() {
788         return getMonth().firstDayOfYear(isLeapYear()) + day - 1;
789     }
790
791     /**
792      * Gets the day-of-week field, which is an enum {@code DayOfWeek}.
793      * <p>
794      * This method returns the enum {@link DayOfWeek} for the day-of-week.
795      * This avoids confusion as to what {@code int} values mean.
796      * If you need access to the primitive {@code int} value then the enum
797      * provides the {@link DayOfWeek#getValue() int value}.
798      * <p>
799      * Additional information can be obtained from the {@code DayOfWeek}.
800      * This includes textual names of the values.
801      *
802      * @return the day-of-week, not null
803      */
804     public DayOfWeek getDayOfWeek() {
805         int dow0 = (int) Math.floorMod(toEpochDay() + 3, 7);
806         return DayOfWeek.of(dow0 + 1);
807     }
808
809     //-----
810     /**
811      * Checks if the year is a leap year, according to the ISO proleptic
812      * calendar system rules.
813      * <p>
814      * This method applies the current rules for leap years across the whole time-line.
815      * In general, a year is a leap year if it is divisible by four without
816      * remainder. However, years divisible by 100, are not leap years, with
817      * the exception of years divisible by 400 which are.
818      * <p>
819      * For example, 1904 is a leap year it is divisible by 4.
820      * 1900 was not a leap year as it is divisible by 100, however 2000 was a
821      * leap year as it is divisible by 400.
822      * <p>
823      * The calculation is proleptic - applying the same rules into the far future and far past.
824      * This is historically inaccurate, but is correct for the ISO-8601 standard.
825      *
826      * @return true if the year is leap, false otherwise
827      */
828     @Override // override for Javadoc and performance
829     public boolean isLeapYear() {

```

```

830         return IsoChronology.INSTANCE.isLeapYear(year);
831     }
832
833     /**
834      * Returns the length of the month represented by this date.
835      * <p>
836      * This returns the length of the month in days.
837      * For example, a date in January would return 31.
838      *
839      * @return the length of the month in days
840      */
841     @Override
842     public int lengthOfMonth() {
843         switch (month) {
844             case 2:
845                 return (isLeapYear() ? 29 : 28);
846             case 4:
847             case 6:
848             case 9:
849             case 11:
850                 return 30;
851             default:
852                 return 31;
853         }
854     }
855
856     /**
857      * Returns the length of the year represented by this date.
858      * <p>
859      * This returns the length of the year in days, either 365 or 366.
860      *
861      * @return 366 if the year is leap, 365 otherwise
862      */
863     @Override // override for Javadoc and performance
864     public int lengthOfYear() {
865         return (isLeapYear() ? 366 : 365);
866     }
867
868     //-----
869     /**
870      * Returns an adjusted copy of this date.
871      * <p>
872      * This returns a {@code LocalDate}, based on this one, with the date adjusted.
873      * The adjustment takes place using the specified adjuster strategy object.
874      * Read the documentation of the adjuster to understand what adjustment will be made.
875      * <p>
876      * A simple adjuster might simply set the one of the fields, such as the year field.
877      * A more complex adjuster might set the date to the last day of the month.
878      * <p>
879      * A selection of common adjustments is provided in
880      * {@link java.time.temporal.TemporalAdjusters TemporalAdjusters}.
881      * These include finding the "last day of the month" and "next Wednesday".
882      * Key date-time classes also implement the {@code TemporalAdjuster} interface,

```



```

883 * such as {@link Month} and {@link java.time.MonthDay MonthDay}.
884 * The adjuster is responsible for handling special cases, such as the varying
885 * lengths of month and leap years.
886 * <p>
887 * For example this code returns a date on the last day of July:
888 * <pre>
889 * import static java.time.Month.*;
890 * import static java.time.temporal.TemporalAdjusters.*;
891 *
892 * result = localDate.with(JULY).with(lastDayOfMonth());
893 * </pre>
894 * <p>
895 * The result of this method is obtained by invoking the
896 * {@link TemporalAdjuster#adjustInto(Temporal)} method on the
897 * specified adjuster passing {@code this} as the argument.
898 * <p>
899 * This instance is immutable and unaffected by this method call.
900 *
901 * @param adjuster the adjuster to use, not null
902 * @return a {@code LocalDate} based on {@code this} with the adjustment made, not null
903 * @throws DateTimeException if the adjustment cannot be made
904 * @throws ArithmeticException if numeric overflow occurs
905 */
906 @Override
907 public LocalDate with(TemporalAdjuster adjuster) {
908     // optimizations
909     if (adjuster instanceof LocalDate) {
910         return (LocalDate) adjuster;
911     }
912     return (LocalDate) adjuster.adjustInto(this);
913 }
914
915 /**
916 * Returns a copy of this date with the specified field set to a new value.
917 * <p>
918 * This returns a {@code LocalDate}, based on this one, with the value
919 * for the specified field changed.
920 * This can be used to change any supported field, such as the year, month or day-of-month.
921 * If it is not possible to set the value, because the field is not supported or for
922 * some other reason, an exception is thrown.
923 * <p>
924 * In some cases, changing the specified field can cause the resulting date to become invalid,
925 * such as changing the month from 31st January to February would make the day-of-month invalid
926 *
927 * In cases like this, the field is responsible for resolving the date. Typically it will
928 * choose
929 * the previous valid date, which would be the last valid day of February in this example.
930 * <p>
931 * If the field is a {@link ChronoField} then the adjustment is implemented here.
932 * The supported fields behave as follows:
933 * <ul>
934 * <li>{@code DAY_OF_WEEK} -
935 * Returns a {@code LocalDate} with the specified day-of-week.

```

```

934 * The date is adjusted up to 6 days forward or backward within the boundary
935 * of a Monday to Sunday week.
936 * <li>{@code ALIGNED_DAY_OF_WEEK_IN_MONTH} -
937 * Returns a {@code LocalDate} with the specified aligned-day-of-week.
938 * The date is adjusted to the specified month-based aligned-day-of-week.
939 * Aligned weeks are counted such that the first week of a given month starts
940 * on the first day of that month.
941 * This may cause the date to be moved up to 6 days into the following month.
942 * <li>{@code ALIGNED_DAY_OF_WEEK_IN_YEAR} -
943 * Returns a {@code LocalDate} with the specified aligned-day-of-week.
944 * The date is adjusted to the specified year-based aligned-day-of-week.
945 * Aligned weeks are counted such that the first week of a given year starts
946 * on the first day of that year.
947 * This may cause the date to be moved up to 6 days into the following year.
948 * <li>{@code DAY_OF_MONTH} -
949 * Returns a {@code LocalDate} with the specified day-of-month.
950 * The month and year will be unchanged. If the day-of-month is invalid for the
951 * year and month, then a {@code DateTimeException} is thrown.
952 * <li>{@code DAY_OF_YEAR} -
953 * Returns a {@code LocalDate} with the specified day-of-year.
954 * The year will be unchanged. If the day-of-year is invalid for the
955 * year, then a {@code DateTimeException} is thrown.
956 * <li>{@code EPOCH_DAY} -
957 * Returns a {@code LocalDate} with the specified epoch-day.
958 * This completely replaces the date and is equivalent to {@link #ofEpochDay(long)}.
959 * <li>{@code ALIGNED_WEEK_OF_MONTH} -
960 * Returns a {@code LocalDate} with the specified aligned-week-of-month.
961 * Aligned weeks are counted such that the first week of a given month starts
962 * on the first day of that month.
963 * This adjustment moves the date in whole week chunks to match the specified week.
964 * The result will have the same day-of-week as this date.
965 * This may cause the date to be moved into the following month.
966 * <li>{@code ALIGNED_WEEK_OF_YEAR} -
967 * Returns a {@code LocalDate} with the specified aligned-week-of-year.
968 * Aligned weeks are counted such that the first week of a given year starts
969 * on the first day of that year.
970 * This adjustment moves the date in whole week chunks to match the specified week.
971 * The result will have the same day-of-week as this date.
972 * This may cause the date to be moved into the following year.
973 * <li>{@code MONTH_OF_YEAR} -
974 * Returns a {@code LocalDate} with the specified month-of-year.
975 * The year will be unchanged. The day-of-month will also be unchanged,
976 * unless it would be invalid for the new month and year. In that case, the
977 * day-of-month is adjusted to the maximum valid value for the new month and year.
978 * <li>{@code PROLEPTIC_MONTH} -
979 * Returns a {@code LocalDate} with the specified proleptic-month.
980 * The day-of-month will be unchanged, unless it would be invalid for the new month
981 * and year. In that case, the day-of-month is adjusted to the maximum valid value
982 * for the new month and year.
983 * <li>{@code YEAR_OF_ERA} -
984 * Returns a {@code LocalDate} with the specified year-of-era.
985 * The era and month will be unchanged. The day-of-month will also be unchanged,
986 * unless it would be invalid for the new month and year. In that case, the

```

```

987 * day-of-month is adjusted to the maximum valid value for the new month and year.
988 * <li>{@code YEAR} -
989 * Returns a {@code LocalDate} with the specified year.
990 * The month will be unchanged. The day-of-month will also be unchanged,
991 * unless it would be invalid for the new month and year. In that case, the
992 * day-of-month is adjusted to the maximum valid value for the new month and year.
993 * </li>{@code ERA} -
994 * Returns a {@code LocalDate} with the specified era.
995 * The year-of-era and month will be unchanged. The day-of-month will also be unchanged,
996 * unless it would be invalid for the new month and year. In that case, the
997 * day-of-month is adjusted to the maximum valid value for the new month and year.
998 * </ul>
999 * <p>
1000 * In all cases, if the new value is outside the valid range of values for the field
1001 * then a {@code DateTimeException} will be thrown.
1002 * <p>
1003 * All other {@code ChronoField} instances will throw an {@code
    UnsupportedTemporalTypeException}.
1004 * <p>
1005 * If the field is not a {@code ChronoField}, then the result of this method
1006 * is obtained by invoking {@code TemporalField.adjustInto(Temporal, long)}
1007 * passing {@code this} as the argument. In this case, the field determines
1008 * whether and how to adjust the instant.
1009 * <p>
1010 * This instance is immutable and unaffected by this method call.
1011 *
1012 * @param field the field to set in the result, not null
1013 * @param newValue the new value of the field in the result
1014 * @return a {@code LocalDate} based on {@code this} with the specified field set, not null
1015 * @throws DateTimeException if the field cannot be set
1016 * @throws UnsupportedTemporalTypeException if the field is not supported
1017 * @throws ArithmeticException if numeric overflow occurs
1018 */
1019 @Override
1020 public LocalDate with(TemporalField field, long newValue) {
1021     if (field instanceof ChronoField) {
1022         ChronoField f = (ChronoField) field;
1023         f.checkValidValue(newValue);
1024         switch (f) {
1025             case DAY_OF_WEEK: return plusDays(newValue - getDayOfWeek().getValue());
1026             case ALIGNED_DAY_OF_WEEK_IN_MONTH: return plusDays(newValue - getLong(
                ALIGNED_DAY_OF_WEEK_IN_MONTH));
1027             case ALIGNED_DAY_OF_WEEK_IN_YEAR: return plusDays(newValue - getLong(
                ALIGNED_DAY_OF_WEEK_IN_YEAR));
1028             case DAY_OF_MONTH: return withDayOfMonth((int) newValue);
1029             case DAY_OF_YEAR: return withDayOfYear((int) newValue);
1030             case EPOCH_DAY: return LocalDate.ofEpochDay(newValue);
1031             case ALIGNED_WEEK_OF_MONTH: return plusWeeks(newValue - getLong(
                ALIGNED_WEEK_OF_MONTH));
1032             case ALIGNED_WEEK_OF_YEAR: return plusWeeks(newValue - getLong(ALIGNED_WEEK_OF_YEAR
                ));
1033             case MONTH_OF_YEAR: return withMonth((int) newValue);
1034             case PROLEPTIC_MONTH: return plusMonths(newValue - getProlepticMonth());

```

```

1035         case YEAR_OF_ERA: return withYear((int) (year >= 1 ? newValue : 1 - newValue));
1036         case YEAR: return withYear((int) newValue);
1037         case ERA: return (getLong(ERA) == newValue ? this : withYear(1 - year));
1038     }
1039     throw new UnsupportedOperationException("Unsupported field: " + field);
1040 }
1041 return field.adjustInto(this, newValue);
1042 }
1043
1044 //-----
1045 /**
1046  * Returns a copy of this {@code LocalDate} with the year altered.
1047  * <p>
1048  * If the day-of-month is invalid for the year, it will be changed to the last valid day of the
1049  * month.
1050  * <p>
1051  * This instance is immutable and unaffected by this method call.
1052  *
1053  * @param year the year to set in the result, from MIN_YEAR to MAX_YEAR
1054  * @return a {@code LocalDate} based on this date with the requested year, not null
1055  * @throws DateTimeException if the year value is invalid
1056  */
1057 public LocalDate withYear(int year) {
1058     if (this.year == year) {
1059         return this;
1060     }
1061     YEAR.checkValidValue(year);
1062     return resolvePreviousValid(year, month, day);
1063 }
1064
1065 /**
1066  * Returns a copy of this {@code LocalDate} with the month-of-year altered.
1067  * <p>
1068  * If the day-of-month is invalid for the year, it will be changed to the last valid day of the
1069  * month.
1070  * <p>
1071  * This instance is immutable and unaffected by this method call.
1072  *
1073  * @param month the month-of-year to set in the result, from 1 (January) to 12 (December)
1074  * @return a {@code LocalDate} based on this date with the requested month, not null
1075  * @throws DateTimeException if the month-of-year value is invalid
1076  */
1077 public LocalDate withMonth(int month) {
1078     if (this.month == month) {
1079         return this;
1080     }
1081     MONTH_OF_YEAR.checkValidValue(month);
1082     return resolvePreviousValid(year, month, day);
1083 }
1084
1085 /**
1086  * Returns a copy of this {@code LocalDate} with the day-of-month altered.
1087  * <p>

```

```

1086     * If the resulting date is invalid, an exception is thrown.
1087     * <p>
1088     * This instance is immutable and unaffected by this method call.
1089     *
1090     * @param dayOfMonth the day-of-month to set in the result, from 1 to 28-31
1091     * @return a {@code LocalDate} based on this date with the requested day, not null
1092     * @throws DateTimeException if the day-of-month value is invalid,
1093     * or if the day-of-month is invalid for the month-year
1094     */
1095     public LocalDate withDayOfMonth(int dayOfMonth) {
1096         if (this.day == dayOfMonth) {
1097             return this;
1098         }
1099         return of(year, month, dayOfMonth);
1100     }
1101
1102     /**
1103     * Returns a copy of this {@code LocalDate} with the day-of-year altered.
1104     * <p>
1105     * If the resulting date is invalid, an exception is thrown.
1106     * <p>
1107     * This instance is immutable and unaffected by this method call.
1108     *
1109     * @param dayOfYear the day-of-year to set in the result, from 1 to 365-366
1110     * @return a {@code LocalDate} based on this date with the requested day, not null
1111     * @throws DateTimeException if the day-of-year value is invalid,
1112     * or if the day-of-year is invalid for the year
1113     */
1114     public LocalDate withDayOfYear(int dayOfYear) {
1115         if (this.getDayOfYear() == dayOfYear) {
1116             return this;
1117         }
1118         return ofYearDay(year, dayOfYear);
1119     }
1120
1121     //-----
1122     /**
1123     * Returns a copy of this date with the specified amount added.
1124     * <p>
1125     * This returns a {@code LocalDate}, based on this one, with the specified amount added.
1126     * The amount is typically {@link Period} but may be any other type implementing
1127     * the {@link TemporalAmount} interface.
1128     * <p>
1129     * The calculation is delegated to the amount object by calling
1130     * {@link TemporalAmount#addTo(Temporal)}. The amount implementation is free
1131     * to implement the addition in any way it wishes, however it typically
1132     * calls back to {@link #plus(long, TemporalUnit)}. Consult the documentation
1133     * of the amount implementation to determine if it can be successfully added.
1134     * <p>
1135     * This instance is immutable and unaffected by this method call.
1136     *
1137     * @param amountToAdd the amount to add, not null
1138     * @return a {@code LocalDate} based on this date with the addition made, not null

```

```

1139     * @throws DateTimeException if the addition cannot be made
1140     * @throws ArithmeticException if numeric overflow occurs
1141     */
1142     @Override
1143     public LocalDate plus(TemporalAmount amountToAdd) {
1144         if (amountToAdd instanceof Period) {
1145             Period periodToAdd = (Period) amountToAdd;
1146             return plusMonths(periodToAdd.toTotalMonths()).plusDays(periodToAdd.getDays());
1147         }
1148         Objects.requireNonNull(amountToAdd, "amountToAdd");
1149         return (LocalDate) amountToAdd.addTo(this);
1150     }
1151
1152     /**
1153      * Returns a copy of this date with the specified amount added.
1154      * <p>
1155      * This returns a {@code LocalDate}, based on this one, with the amount
1156      * in terms of the unit added. If it is not possible to add the amount, because the
1157      * unit is not supported or for some other reason, an exception is thrown.
1158      * <p>
1159      * In some cases, adding the amount can cause the resulting date to become invalid.
1160      * For example, adding one month to 31st January would result in 31st February.
1161      * In cases like this, the unit is responsible for resolving the date.
1162      * Typically it will choose the previous valid date, which would be the last valid
1163      * day of February in this example.
1164      * <p>
1165      * If the field is a {@link ChronoUnit} then the addition is implemented here.
1166      * The supported fields behave as follows:
1167      * <ul>
1168      * <li>{@code DAYS} -
1169      * Returns a {@code LocalDate} with the specified number of days added.
1170      * This is equivalent to {@link #plusDays(long)}.
1171      * <li>{@code WEEKS} -
1172      * Returns a {@code LocalDate} with the specified number of weeks added.
1173      * This is equivalent to {@link #plusWeeks(long)} and uses a 7 day week.
1174      * <li>{@code MONTHS} -
1175      * Returns a {@code LocalDate} with the specified number of months added.
1176      * This is equivalent to {@link #plusMonths(long)}.
1177      * The day-of-month will be unchanged unless it would be invalid for the new
1178      * month and year. In that case, the day-of-month is adjusted to the maximum
1179      * valid value for the new month and year.
1180      * <li>{@code YEARS} -
1181      * Returns a {@code LocalDate} with the specified number of years added.
1182      * This is equivalent to {@link #plusYears(long)}.
1183      * The day-of-month will be unchanged unless it would be invalid for the new
1184      * month and year. In that case, the day-of-month is adjusted to the maximum
1185      * valid value for the new month and year.
1186      * <li>{@code DECADES} -
1187      * Returns a {@code LocalDate} with the specified number of decades added.
1188      * This is equivalent to calling {@link #plusYears(long)} with the amount
1189      * multiplied by 10.
1190      * The day-of-month will be unchanged unless it would be invalid for the new
1191      * month and year. In that case, the day-of-month is adjusted to the maximum

```

```

1192 * valid value for the new month and year.
1193 * <li>{@code CENTURIES} -
1194 * Returns a {@code LocalDate} with the specified number of centuries added.
1195 * This is equivalent to calling {@link #plusYears(long)} with the amount
1196 * multiplied by 100.
1197 * The day-of-month will be unchanged unless it would be invalid for the new
1198 * month and year. In that case, the day-of-month is adjusted to the maximum
1199 * valid value for the new month and year.
1200 * <li>{@code MILLENNIA} -
1201 * Returns a {@code LocalDate} with the specified number of millennia added.
1202 * This is equivalent to calling {@link #plusYears(long)} with the amount
1203 * multiplied by 1,000.
1204 * The day-of-month will be unchanged unless it would be invalid for the new
1205 * month and year. In that case, the day-of-month is adjusted to the maximum
1206 * valid value for the new month and year.
1207 * <li>{@code ERAS} -
1208 * Returns a {@code LocalDate} with the specified number of eras added.
1209 * Only two eras are supported so the amount must be one, zero or minus one.
1210 * If the amount is non-zero then the year is changed such that the year-of-era
1211 * is unchanged.
1212 * The day-of-month will be unchanged unless it would be invalid for the new
1213 * month and year. In that case, the day-of-month is adjusted to the maximum
1214 * valid value for the new month and year.
1215 * </ul>
1216 * <p>
1217 * All other {@code ChronoUnit} instances will throw an {@code UnsupportedOperationException}
1218 *   }.
1219 * <p>
1220 * If the field is not a {@code ChronoUnit}, then the result of this method
1221 * is obtained by invoking {@code TemporalUnit.addTo(Temporal, long)}
1222 * passing {@code this} as the argument. In this case, the unit determines
1223 * whether and how to perform the addition.
1224 * <p>
1225 * This instance is immutable and unaffected by this method call.
1226 *
1227 * @param amountToAdd the amount of the unit to add to the result, may be negative
1228 * @param unit the unit of the amount to add, not null
1229 * @return a {@code LocalDate} based on this date with the specified amount added, not null
1230 * @throws DateTimeException if the addition cannot be made
1231 * @throws UnsupportedOperationException if the unit is not supported
1232 * @throws ArithmeticException if numeric overflow occurs
1233 */
1234 @Override
1235 public LocalDate plus(long amountToAdd, TemporalUnit unit) {
1236     if (unit instanceof ChronoUnit) {
1237         ChronoUnit f = (ChronoUnit) unit;
1238         switch (f) {
1239             case DAYS: return plusDays(amountToAdd);
1240             case WEEKS: return plusWeeks(amountToAdd);
1241             case MONTHS: return plusMonths(amountToAdd);
1242             case YEARS: return plusYears(amountToAdd);
1243             case DECADES: return plusYears(Math.multiplyExact(amountToAdd, 10));
1244             case CENTURIES: return plusYears(Math.multiplyExact(amountToAdd, 100));

```



```

1244         case MILLENNIA: return plusYears(Math.multiplyExact(amountToAdd, 1000));
1245         case ERAS: return with(ERA, Math.addExact(getLong(ERA), amountToAdd));
1246     }
1247     throw new UnsupportedOperationException("Unsupported unit: " + unit);
1248 }
1249 return unit.addTo(this, amountToAdd);
1250 }
1251
1252 //-----
1253 /**
1254  * Returns a copy of this {@code LocalDate} with the specified number of years added.
1255  * <p>
1256  * This method adds the specified amount to the years field in three steps:
1257  * <ol>
1258  * <li>Add the input years to the year field</li>
1259  * <li>Check if the resulting date would be invalid</li>
1260  * <li>Adjust the day-of-month to the last valid day if necessary</li>
1261  * </ol>
1262  * <p>
1263  * For example, 2008-02-29 (leap year) plus one year would result in the
1264  * invalid date 2009-02-29 (standard year). Instead of returning an invalid
1265  * result, the last valid day of the month, 2009-02-28, is selected instead.
1266  * <p>
1267  * This instance is immutable and unaffected by this method call.
1268  *
1269  * @param yearsToAdd the years to add, may be negative
1270  * @return a {@code LocalDate} based on this date with the years added, not null
1271  * @throws DateTimeException if the result exceeds the supported date range
1272  */
1273 public LocalDate plusYears(long yearsToAdd) {
1274     if (yearsToAdd == 0) {
1275         return this;
1276     }
1277     int newYear = YEAR.checkValidIntValue(year + yearsToAdd); // safe overflow
1278     return resolvePreviousValid(newYear, month, day);
1279 }
1280
1281 /**
1282  * Returns a copy of this {@code LocalDate} with the specified number of months added.
1283  * <p>
1284  * This method adds the specified amount to the months field in three steps:
1285  * <ol>
1286  * <li>Add the input months to the month-of-year field</li>
1287  * <li>Check if the resulting date would be invalid</li>
1288  * <li>Adjust the day-of-month to the last valid day if necessary</li>
1289  * </ol>
1290  * <p>
1291  * For example, 2007-03-31 plus one month would result in the invalid date
1292  * 2007-04-31. Instead of returning an invalid result, the last valid day
1293  * of the month, 2007-04-30, is selected instead.
1294  * <p>
1295  * This instance is immutable and unaffected by this method call.
1296  *

```



```

1297     * @param monthsToAdd the months to add, may be negative
1298     * @return a {@code LocalDate} based on this date with the months added, not null
1299     * @throws DateTimeException if the result exceeds the supported date range
1300     */
1301     public LocalDate plusMonths(long monthsToAdd) {
1302         if (monthsToAdd == 0) {
1303             return this;
1304         }
1305         long monthCount = year * 12L + (month - 1);
1306         long calcMonths = monthCount + monthsToAdd; // safe overflow
1307         int newYear = YEAR.checkValidIntValue(Math.floorDiv(calcMonths, 12));
1308         int newMonth = (int)Math.floorMod(calcMonths, 12) + 1;
1309         return resolvePreviousValid(newYear, newMonth, day);
1310     }
1311
1312     /**
1313     * Returns a copy of this {@code LocalDate} with the specified number of weeks added.
1314     * <p>
1315     * This method adds the specified amount in weeks to the days field incrementing
1316     * the month and year fields as necessary to ensure the result remains valid.
1317     * The result is only invalid if the maximum/minimum year is exceeded.
1318     * <p>
1319     * For example, 2008-12-31 plus one week would result in 2009-01-07.
1320     * <p>
1321     * This instance is immutable and unaffected by this method call.
1322     *
1323     * @param weeksToAdd the weeks to add, may be negative
1324     * @return a {@code LocalDate} based on this date with the weeks added, not null
1325     * @throws DateTimeException if the result exceeds the supported date range
1326     */
1327     public LocalDate plusWeeks(long weeksToAdd) {
1328         return plusDays(Math.multiplyExact(weeksToAdd, 7));
1329     }
1330
1331     /**
1332     * Returns a copy of this {@code LocalDate} with the specified number of days added.
1333     * <p>
1334     * This method adds the specified amount to the days field incrementing the
1335     * month and year fields as necessary to ensure the result remains valid.
1336     * The result is only invalid if the maximum/minimum year is exceeded.
1337     * <p>
1338     * For example, 2008-12-31 plus one day would result in 2009-01-01.
1339     * <p>
1340     * This instance is immutable and unaffected by this method call.
1341     *
1342     * @param daysToAdd the days to add, may be negative
1343     * @return a {@code LocalDate} based on this date with the days added, not null
1344     * @throws DateTimeException if the result exceeds the supported date range
1345     */
1346     public LocalDate plusDays(long daysToAdd) {
1347         if (daysToAdd == 0) {
1348             return this;
1349         }

```

```

1350     long mjdDay = Math.addExact(toEpochDay(), daysToAdd);
1351     return LocalDate.ofEpochDay(mjdDay);
1352 }
1353
1354 //-----
1355 /**
1356  * Returns a copy of this date with the specified amount subtracted.
1357  * <p>
1358  * This returns a {@code LocalDate}, based on this one, with the specified amount subtracted.
1359  * The amount is typically {@link Period} but may be any other type implementing
1360  * the {@link TemporalAmount} interface.
1361  * <p>
1362  * The calculation is delegated to the amount object by calling
1363  * {@link TemporalAmount#subtractFrom(Temporal)}. The amount implementation is free
1364  * to implement the subtraction in any way it wishes, however it typically
1365  * calls back to {@link #minus(long, TemporalUnit)}. Consult the documentation
1366  * of the amount implementation to determine if it can be successfully subtracted.
1367  * <p>
1368  * This instance is immutable and unaffected by this method call.
1369  *
1370  * @param amountToSubtract the amount to subtract, not null
1371  * @return a {@code LocalDate} based on this date with the subtraction made, not null
1372  * @throws DateTimeException if the subtraction cannot be made
1373  * @throws ArithmeticException if numeric overflow occurs
1374  */
1375 @Override
1376 public LocalDate minus(TemporalAmount amountToSubtract) {
1377     if (amountToSubtract instanceof Period) {
1378         Period periodToSubtract = (Period) amountToSubtract;
1379         return minusMonths(periodToSubtract.getTotalMonths()).minusDays(periodToSubtract.getDays());
1380     }
1381     Objects.requireNonNull(amountToSubtract, "amountToSubtract");
1382     return (LocalDate) amountToSubtract.subtractFrom(this);
1383 }
1384
1385 /**
1386  * Returns a copy of this date with the specified amount subtracted.
1387  * <p>
1388  * This returns a {@code LocalDate}, based on this one, with the amount
1389  * in terms of the unit subtracted. If it is not possible to subtract the amount,
1390  * because the unit is not supported or for some other reason, an exception is thrown.
1391  * <p>
1392  * This method is equivalent to {@link #plus(long, TemporalUnit)} with the amount negated.
1393  * See that method for a full description of how addition, and thus subtraction, works.
1394  * <p>
1395  * This instance is immutable and unaffected by this method call.
1396  *
1397  * @param amountToSubtract the amount of the unit to subtract from the result, may be negative
1398  * @param unit the unit of the amount to subtract, not null
1399  * @return a {@code LocalDate} based on this date with the specified amount subtracted, not
1400         null
1401  * @throws DateTimeException if the subtraction cannot be made

```

```

1401     * @throws UnsupportedOperationException if the unit is not supported
1402     * @throws ArithmeticException if numeric overflow occurs
1403     */
1404     @Override
1405     public LocalDate minus(long amountToSubtract, TemporalUnit unit) {
1406         return (amountToSubtract == Long.MIN_VALUE ? plus(Long.MAX_VALUE, unit).plus(1, unit) :
1407             plus(-amountToSubtract, unit));
1408     }
1409
1410     //-----
1411     /**
1412      * Returns a copy of this {@code LocalDate} with the specified number of years subtracted.
1413      * <p>
1414      * This method subtracts the specified amount from the years field in three steps:
1415      * <ol>
1416      * <li>Subtract the input years from the year field</li>
1417      * <li>Check if the resulting date would be invalid</li>
1418      * <li>Adjust the day-of-month to the last valid day if necessary</li>
1419      * </ol>
1420      * <p>
1421      * For example, 2008-02-29 (leap year) minus one year would result in the
1422      * invalid date 2007-02-29 (standard year). Instead of returning an invalid
1423      * result, the last valid day of the month, 2007-02-28, is selected instead.
1424      * <p>
1425      * This instance is immutable and unaffected by this method call.
1426      *
1427      * @param yearsToSubtract the years to subtract, may be negative
1428      * @return a {@code LocalDate} based on this date with the years subtracted, not null
1429      * @throws DateTimeException if the result exceeds the supported date range
1430      */
1431     public LocalDate minusYears(long yearsToSubtract) {
1432         return (yearsToSubtract == Long.MIN_VALUE ? plusYears(Long.MAX_VALUE).plusYears(1) :
1433             plusYears(-yearsToSubtract));
1434     }
1435
1436     /**
1437      * Returns a copy of this {@code LocalDate} with the specified number of months subtracted.
1438      * <p>
1439      * This method subtracts the specified amount from the months field in three steps:
1440      * <ol>
1441      * <li>Subtract the input months from the month-of-year field</li>
1442      * <li>Check if the resulting date would be invalid</li>
1443      * <li>Adjust the day-of-month to the last valid day if necessary</li>
1444      * </ol>
1445      * <p>
1446      * For example, 2007-03-31 minus one month would result in the invalid date
1447      * 2007-02-31. Instead of returning an invalid result, the last valid day
1448      * of the month, 2007-02-28, is selected instead.
1449      * <p>
1450      * This instance is immutable and unaffected by this method call.
1451      *
1452      * @param monthsToSubtract the months to subtract, may be negative
1453      * @return a {@code LocalDate} based on this date with the months subtracted, not null

```

```

1452     * @throws DateTimeException if the result exceeds the supported date range
1453     */
1454     public LocalDate minusMonths(long monthsToSubtract) {
1455         return (monthsToSubtract == Long.MIN_VALUE ? plusMonths(Long.MAX_VALUE).plusMonths(1) :
1456             plusMonths(-monthsToSubtract));
1457     }
1458     /**
1459     * Returns a copy of this {@code LocalDate} with the specified number of weeks subtracted.
1460     * <p>
1461     * This method subtracts the specified amount in weeks from the days field decrementing
1462     * the month and year fields as necessary to ensure the result remains valid.
1463     * The result is only invalid if the maximum/minimum year is exceeded.
1464     * <p>
1465     * For example, 2009-01-07 minus one week would result in 2008-12-31.
1466     * <p>
1467     * This instance is immutable and unaffected by this method call.
1468     *
1469     * @param weeksToSubtract the weeks to subtract, may be negative
1470     * @return a {@code LocalDate} based on this date with the weeks subtracted, not null
1471     * @throws DateTimeException if the result exceeds the supported date range
1472     */
1473     public LocalDate minusWeeks(long weeksToSubtract) {
1474         return (weeksToSubtract == Long.MIN_VALUE ? plusWeeks(Long.MAX_VALUE).plusWeeks(1) :
1475             plusWeeks(-weeksToSubtract));
1476     }
1477     /**
1478     * Returns a copy of this {@code LocalDate} with the specified number of days subtracted.
1479     * <p>
1480     * This method subtracts the specified amount from the days field decrementing the
1481     * month and year fields as necessary to ensure the result remains valid.
1482     * The result is only invalid if the maximum/minimum year is exceeded.
1483     * <p>
1484     * For example, 2009-01-01 minus one day would result in 2008-12-31.
1485     * <p>
1486     * This instance is immutable and unaffected by this method call.
1487     *
1488     * @param daysToSubtract the days to subtract, may be negative
1489     * @return a {@code LocalDate} based on this date with the days subtracted, not null
1490     * @throws DateTimeException if the result exceeds the supported date range
1491     */
1492     public LocalDate minusDays(long daysToSubtract) {
1493         return (daysToSubtract == Long.MIN_VALUE ? plusDays(Long.MAX_VALUE).plusDays(1) : plusDays
1494             (-daysToSubtract));
1495     }
1496     //-----
1497     /**
1498     * Queries this date using the specified query.
1499     * <p>
1500     * This queries this date using the specified query strategy object.
1501     * The {@code TemporalQuery} object defines the logic to be used to

```

```

1502     * obtain the result. Read the documentation of the query to understand
1503     * what the result of this method will be.
1504     * <p>
1505     * The result of this method is obtained by invoking the
1506     * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
1507     * specified query passing {@code this} as the argument.
1508     *
1509     * @param <R> the type of the result
1510     * @param query the query to invoke, not null
1511     * @return the query result, null may be returned (defined by the query)
1512     * @throws DateTimeException if unable to query (defined by the query)
1513     * @throws ArithmeticException if numeric overflow occurs (defined by the query)
1514     */
1515     @SuppressWarnings("unchecked")
1516     @Override
1517     public <R> R query(TemporalQuery<R> query) {
1518         if (query == TemporalQueries.localDate()) {
1519             return (R) this;
1520         }
1521         return ChronoLocalDate.super.query(query);
1522     }
1523
1524     /**
1525     * Adjusts the specified temporal object to have the same date as this object.
1526     * <p>
1527     * This returns a temporal object of the same observable type as the input
1528     * with the date changed to be the same as this.
1529     * <p>
1530     * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
1531     * passing {@link ChronoField#EPOCH_DAY} as the field.
1532     * <p>
1533     * In most cases, it is clearer to reverse the calling pattern by using
1534     * {@link Temporal#with(TemporalAdjuster)}:
1535     * <pre>
1536     * // these two lines are equivalent, but the second approach is recommended
1537     * temporal = thisLocalDate.adjustInto(temporal);
1538     * temporal = temporal.with(thisLocalDate);
1539     * </pre>
1540     * <p>
1541     * This instance is immutable and unaffected by this method call.
1542     *
1543     * @param temporal the target object to be adjusted, not null
1544     * @return the adjusted object, not null
1545     * @throws DateTimeException if unable to make the adjustment
1546     * @throws ArithmeticException if numeric overflow occurs
1547     */
1548     @Override // override for Javadoc
1549     public Temporal adjustInto(Temporal temporal) {
1550         return ChronoLocalDate.super.adjustInto(temporal);
1551     }
1552
1553     /**
1554     * Calculates the amount of time until another date in terms of the specified unit.

```

```

1555 * <p>
1556 * This calculates the amount of time between two {@code LocalDate}
1557 * objects in terms of a single {@code TemporalUnit}.
1558 * The start and end points are {@code this} and the specified date.
1559 * The result will be negative if the end is before the start.
1560 * The {@code Temporal} passed to this method is converted to a
1561 * {@code LocalDate} using {@link #from(TemporalAccessor)}.
1562 * For example, the amount in days between two dates can be calculated
1563 * using {@code startDate.until(endDate, DAYS)}.
1564 * <p>
1565 * The calculation returns a whole number, representing the number of
1566 * complete units between the two dates.
1567 * For example, the amount in months between 2012-06-15 and 2012-08-14
1568 * will only be one month as it is one day short of two months.
1569 * <p>
1570 * There are two equivalent ways of using this method.
1571 * The first is to invoke this method.
1572 * The second is to use {@link TemporalUnit#between(Temporal, Temporal)}:
1573 * <pre>
1574 *     // these two lines are equivalent
1575 *     amount = start.until(end, MONTHS);
1576 *     amount = MONTHS.between(start, end);
1577 * </pre>
1578 * The choice should be made based on which makes the code more readable.
1579 * <p>
1580 * The calculation is implemented in this method for {@link ChronoUnit}.
1581 * The units {@code DAYS}, {@code WEEKS}, {@code MONTHS}, {@code YEARS},
1582 * {@code DECADES}, {@code CENTURIES}, {@code MILLENNIA} and {@code ERAS}
1583 * are supported. Other {@code ChronoUnit} values will throw an exception.
1584 * <p>
1585 * If the unit is not a {@code ChronoUnit}, then the result of this method
1586 * is obtained by invoking {@code TemporalUnit.between(Temporal, Temporal)}
1587 * passing {@code this} as the first argument and the converted input temporal
1588 * as the second argument.
1589 * <p>
1590 * This instance is immutable and unaffected by this method call.
1591 *
1592 * @param endExclusive the end date, exclusive, which is converted to a {@code LocalDate}, not
1593 *     null
1594 * @param unit the unit to measure the amount in, not null
1595 * @return the amount of time between this date and the end date
1596 * @throws DateTimeException if the amount cannot be calculated, or the end
1597 *     temporal cannot be converted to a {@code LocalDate}
1598 * @throws UnsupportedTemporalTypeException if the unit is not supported
1599 * @throws ArithmeticException if numeric overflow occurs
1600 */
1601 @Override
1602 public long until(Temporal endExclusive, TemporalUnit unit) {
1603     LocalDate end = LocalDate.from(endExclusive);
1604     if (unit instanceof ChronoUnit) {
1605         switch ((ChronoUnit) unit) {
1606             case DAYS: return daysUntil(end);
1607             case WEEKS: return daysUntil(end) / 7;

```

```

1607         case MONTHS: return monthsUntil(end);
1608         case YEARS: return monthsUntil(end) / 12;
1609         case DECADES: return monthsUntil(end) / 120;
1610         case CENTURIES: return monthsUntil(end) / 1200;
1611         case MILLENNIA: return monthsUntil(end) / 12000;
1612         case ERAS: return end.getLong(ERA) - getLong(ERA);
1613     }
1614     throw new UnsupportedOperationException("Unsupported unit: " + unit);
1615 }
1616 return unit.between(this, end);
1617 }
1618
1619 long daysUntil(LocalDate end) {
1620     return end.toEpochDay() - toEpochDay(); // no overflow
1621 }
1622
1623 private long monthsUntil(LocalDate end) {
1624     long packed1 = getProlepticMonth() * 32L + getDayOfMonth(); // no overflow
1625     long packed2 = end.getProlepticMonth() * 32L + end.getDayOfMonth(); // no overflow
1626     return (packed2 - packed1) / 32;
1627 }
1628
1629 /**
1630  * Calculates the period between this date and another date as a {@code Period}.
1631  * <p>
1632  * This calculates the period between two dates in terms of years, months and days.
1633  * The start and end points are {@code this} and the specified date.
1634  * The result will be negative if the end is before the start.
1635  * The negative sign will be the same in each of year, month and day.
1636  * <p>
1637  * The calculation is performed using the ISO calendar system.
1638  * If necessary, the input date will be converted to ISO.
1639  * <p>
1640  * The start date is included, but the end date is not.
1641  * The period is calculated by removing complete months, then calculating
1642  * the remaining number of days, adjusting to ensure that both have the same sign.
1643  * The number of months is then normalized into years and months based on a 12 month year.
1644  * A month is considered to be complete if the end day-of-month is greater
1645  * than or equal to the start day-of-month.
1646  * For example, from {@code 2010-01-15} to {@code 2011-03-18} is "1 year, 2 months and 3 days".
1647  * <p>
1648  * There are two equivalent ways of using this method.
1649  * The first is to invoke this method.
1650  * The second is to use {@link Period#between(LocalDate, LocalDate)}:
1651  * <pre>
1652  *     // these two lines are equivalent
1653  *     period = start.until(end);
1654  *     period = Period.between(start, end);
1655  * </pre>
1656  * The choice should be made based on which makes the code more readable.
1657  *
1658  * @param endDateExclusive the end date, exclusive, which may be in any chronology, not null
1659  * @return the period between this date and the end date, not null

```



```

1660     */
1661     @Override
1662     public Period until(ChronoLocalDate endDateExclusive) {
1663         LocalDate end = LocalDate.from(endDateExclusive);
1664         long totalMonths = end.getProlepticMonth() - this.getProlepticMonth(); // safe
1665         int days = end.day - this.day;
1666         if (totalMonths > 0 && days < 0) {
1667             totalMonths--;
1668             LocalDate calcDate = this.plusMonths(totalMonths);
1669             days = (int) (end.toEpochDay() - calcDate.toEpochDay()); // safe
1670         } else if (totalMonths < 0 && days > 0) {
1671             totalMonths++;
1672             days -= end.lengthOfMonth();
1673         }
1674         long years = totalMonths / 12; // safe
1675         int months = (int) (totalMonths % 12); // safe
1676         return Period.of(Math.toIntExact(years), months, days);
1677     }
1678
1679     /**
1680      * Formats this date using the specified formatter.
1681      * <p>
1682      * This date will be passed to the formatter to produce a string.
1683      *
1684      * @param formatter the formatter to use, not null
1685      * @return the formatted date string, not null
1686      * @throws DateTimeException if an error occurs during printing
1687      */
1688     @Override // override for Javadoc and performance
1689     public String format(DateTimeFormatter formatter) {
1690         Objects.requireNonNull(formatter, "formatter");
1691         return formatter.format(this);
1692     }
1693
1694     //-----
1695     /**
1696      * Combines this date with a time to create a {@code LocalDateTime}.
1697      * <p>
1698      * This returns a {@code LocalDateTime} formed from this date at the specified time.
1699      * All possible combinations of date and time are valid.
1700      *
1701      * @param time the time to combine with, not null
1702      * @return the local date-time formed from this date and the specified time, not null
1703      */
1704     @Override
1705     public LocalDateTime atTime(LocalTime time) {
1706         return LocalDateTime.of(this, time);
1707     }
1708
1709     /**
1710      * Combines this date with a time to create a {@code LocalDateTime}.
1711      * <p>
1712      * This returns a {@code LocalDateTime} formed from this date at the

```



```

1713     * specified hour and minute.
1714     * The seconds and nanosecond fields will be set to zero.
1715     * The individual time fields must be within their valid range.
1716     * All possible combinations of date and time are valid.
1717     *
1718     * @param hour    the hour-of-day to use, from 0 to 23
1719     * @param minute  the minute-of-hour to use, from 0 to 59
1720     * @return the local date-time formed from this date and the specified time, not null
1721     * @throws DateTimeException if the value of any field is out of range
1722     */
1723     public LocalDateTime atTime(int hour, int minute) {
1724         return atTime(LocalTime.of(hour, minute));
1725     }
1726
1727     /**
1728     * Combines this date with a time to create a {@code LocalDateTime}.
1729     * <p>
1730     * This returns a {@code LocalDateTime} formed from this date at the
1731     * specified hour, minute and second.
1732     * The nanosecond field will be set to zero.
1733     * The individual time fields must be within their valid range.
1734     * All possible combinations of date and time are valid.
1735     *
1736     * @param hour    the hour-of-day to use, from 0 to 23
1737     * @param minute  the minute-of-hour to use, from 0 to 59
1738     * @param second  the second-of-minute to represent, from 0 to 59
1739     * @return the local date-time formed from this date and the specified time, not null
1740     * @throws DateTimeException if the value of any field is out of range
1741     */
1742     public LocalDateTime atTime(int hour, int minute, int second) {
1743         return atTime(LocalTime.of(hour, minute, second));
1744     }
1745
1746     /**
1747     * Combines this date with a time to create a {@code LocalDateTime}.
1748     * <p>
1749     * This returns a {@code LocalDateTime} formed from this date at the
1750     * specified hour, minute, second and nanosecond.
1751     * The individual time fields must be within their valid range.
1752     * All possible combinations of date and time are valid.
1753     *
1754     * @param hour    the hour-of-day to use, from 0 to 23
1755     * @param minute  the minute-of-hour to use, from 0 to 59
1756     * @param second  the second-of-minute to represent, from 0 to 59
1757     * @param nanoOfSecond  the nano-of-second to represent, from 0 to 999,999,999
1758     * @return the local date-time formed from this date and the specified time, not null
1759     * @throws DateTimeException if the value of any field is out of range
1760     */
1761     public LocalDateTime atTime(int hour, int minute, int second, int nanoOfSecond) {
1762         return atTime(LocalTime.of(hour, minute, second, nanoOfSecond));
1763     }
1764
1765     /**

```

```

1766     * Combines this date with an offset time to create an {@code OffsetDateTime}.
1767     * <p>
1768     * This returns an {@code OffsetDateTime} formed from this date at the specified time.
1769     * All possible combinations of date and time are valid.
1770     *
1771     * @param time the time to combine with, not null
1772     * @return the offset date-time formed from this date and the specified time, not null
1773     */
1774     public OffsetDateTime atTime(OffsetTime time) {
1775         return OffsetDateTime.of(LocalDateTime.of(this, time.toLocalTime()), time.getOffset());
1776     }
1777
1778     /**
1779     * Combines this date with the time of midnight to create a {@code LocalDateTime}
1780     * at the start of this date.
1781     * <p>
1782     * This returns a {@code LocalDateTime} formed from this date at the time of
1783     * midnight, 00:00, at the start of this date.
1784     *
1785     * @return the local date-time of midnight at the start of this date, not null
1786     */
1787     public LocalDateTime atStartOfDay() {
1788         return LocalDateTime.of(this, LocalTime.MIDNIGHT);
1789     }
1790
1791     /**
1792     * Returns a zoned date-time from this date at the earliest valid time according
1793     * to the rules in the time-zone.
1794     * <p>
1795     * Time-zone rules, such as daylight savings, mean that not every local date-time
1796     * is valid for the specified zone, thus the local date-time may not be midnight.
1797     * <p>
1798     * In most cases, there is only one valid offset for a local date-time.
1799     * In the case of an overlap, there are two valid offsets, and the earlier one is used,
1800     * corresponding to the first occurrence of midnight on the date.
1801     * In the case of a gap, the zoned date-time will represent the instant just after the gap.
1802     * <p>
1803     * If the zone ID is a {@link ZoneOffset}, then the result always has a time of midnight.
1804     * <p>
1805     * To convert to a specific time in a given time-zone call {@link #atTime(LocalTime)}
1806     * followed by {@link LocalDateTime#atZone(ZoneId)}.
1807     *
1808     * @param zone the zone ID to use, not null
1809     * @return the zoned date-time formed from this date and the earliest valid time for the zone,
1810     *         not null
1811     */
1812     public ZonedDateTime atStartOfDay(ZoneId zone) {
1813         Objects.requireNonNull(zone, "zone");
1814         // need to handle case where there is a gap from 11:30 to 00:30
1815         // standard ZDT factory would result in 01:00 rather than 00:30
1816         LocalDateTime ldt = atTime(LocalTime.MIDNIGHT);
1817         if (zone instanceof ZoneOffset == false) {
1818             ZoneRules rules = zone.getRules();

```

```

1818         ZoneOffsetTransition trans = rules.getTransition(ldt);
1819         if (trans != null && trans.isGap()) {
1820             ldt = trans.getDateTimeAfter();
1821         }
1822     }
1823     return ZonedDateTime.of(ldt, zone);
1824 }
1825
1826 //-----
1827 @Override
1828 public long toEpochDay() {
1829     long y = year;
1830     long m = month;
1831     long total = 0;
1832     total += 365 * y;
1833     if (y >= 0) {
1834         total += (y + 3) / 4 - (y + 99) / 100 + (y + 399) / 400;
1835     } else {
1836         total -= y / -4 - y / -100 + y / -400;
1837     }
1838     total += ((367 * m - 362) / 12);
1839     total += day - 1;
1840     if (m > 2) {
1841         total--;
1842         if (isLeapYear() == false) {
1843             total--;
1844         }
1845     }
1846     return total - DAYS_0000_TO_1970;
1847 }
1848
1849 //-----
1850 /**
1851  * Compares this date to another date.
1852  * <p>
1853  * The comparison is primarily based on the date, from earliest to latest.
1854  * It is "consistent with equals", as defined by {@link Comparable}.
1855  * <p>
1856  * If all the dates being compared are instances of {@code LocalDate},
1857  * then the comparison will be entirely based on the date.
1858  * If some dates being compared are in different chronologies, then the
1859  * chronology is also considered, see {@link java.time.chrono.ChronoLocalDate#compareTo}.
1860  *
1861  * @param other the other date to compare to, not null
1862  * @return the comparator value, negative if less, positive if greater
1863  */
1864 @Override // override for Javadoc and performance
1865 public int compareTo(ChronoLocalDate other) {
1866     if (other instanceof LocalDate) {
1867         return compareTo((LocalDate) other);
1868     }
1869     return ChronoLocalDate.super.compareTo(other);
1870 }

```

```

1871
1872     int compareTo(LocalDate otherDate) {
1873         int cmp = (year - otherDate.year);
1874         if (cmp == 0) {
1875             cmp = (month - otherDate.month);
1876             if (cmp == 0) {
1877                 cmp = (day - otherDate.day);
1878             }
1879         }
1880         return cmp;
1881     }
1882
1883     /**
1884      * Checks if this date is after the specified date.
1885      * <p>
1886      * This checks to see if this date represents a point on the
1887      * local time-line after the other date.
1888      * <pre>
1889      *     LocalDate a = LocalDate.of(2012, 6, 30);
1890      *     LocalDate b = LocalDate.of(2012, 7, 1);
1891      *     a.isAfter(b) == false
1892      *     a.isAfter(a) == false
1893      *     b.isAfter(a) == true
1894      * </pre>
1895      * <p>
1896      * This method only considers the position of the two dates on the local time-line.
1897      * It does not take into account the chronology, or calendar system.
1898      * This is different from the comparison in {@link #compareTo(ChronoLocalDate)},
1899      * but is the same approach as {@link ChronoLocalDate#timeLineOrder()}.
1900      *
1901      * @param other the other date to compare to, not null
1902      * @return true if this date is after the specified date
1903      */
1904     @Override // override for Javadoc and performance
1905     public boolean isAfter(ChronoLocalDate other) {
1906         if (other instanceof LocalDate) {
1907             return compareTo((LocalDate) other) > 0;
1908         }
1909         return ChronoLocalDate.super.isAfter(other);
1910     }
1911
1912     /**
1913      * Checks if this date is before the specified date.
1914      * <p>
1915      * This checks to see if this date represents a point on the
1916      * local time-line before the other date.
1917      * <pre>
1918      *     LocalDate a = LocalDate.of(2012, 6, 30);
1919      *     LocalDate b = LocalDate.of(2012, 7, 1);
1920      *     a.isBefore(b) == true
1921      *     a.isBefore(a) == false
1922      *     b.isBefore(a) == false
1923      * </pre>

```

```

1924 * <p>
1925 * This method only considers the position of the two dates on the local time-line.
1926 * It does not take into account the chronology, or calendar system.
1927 * This is different from the comparison in {@link #compareTo(ChronoLocalDate)},
1928 * but is the same approach as {@link ChronoLocalDate#timeLineOrder()}.
1929 *
1930 * @param other the other date to compare to, not null
1931 * @return true if this date is before the specified date
1932 */
1933 @Override // override for Javadoc and performance
1934 public boolean isBefore(ChronoLocalDate other) {
1935     if (other instanceof LocalDate) {
1936         return compareTo((LocalDate) other) < 0;
1937     }
1938     return ChronoLocalDate.super.isBefore(other);
1939 }
1940
1941 /**
1942 * Checks if this date is equal to the specified date.
1943 * <p>
1944 * This checks to see if this date represents the same point on the
1945 * local time-line as the other date.
1946 * <pre>
1947 *     LocalDate a = LocalDate.of(2012, 6, 30);
1948 *     LocalDate b = LocalDate.of(2012, 7, 1);
1949 *     a.isEqual(b) == false
1950 *     a.isEqual(a) == true
1951 *     b.isEqual(a) == false
1952 * </pre>
1953 * <p>
1954 * This method only considers the position of the two dates on the local time-line.
1955 * It does not take into account the chronology, or calendar system.
1956 * This is different from the comparison in {@link #compareTo(ChronoLocalDate)}
1957 * but is the same approach as {@link ChronoLocalDate#timeLineOrder()}.
1958 *
1959 * @param other the other date to compare to, not null
1960 * @return true if this date is equal to the specified date
1961 */
1962 @Override // override for Javadoc and performance
1963 public boolean isEqual(ChronoLocalDate other) {
1964     if (other instanceof LocalDate) {
1965         return compareTo((LocalDate) other) == 0;
1966     }
1967     return ChronoLocalDate.super.isEqual(other);
1968 }
1969
1970 //-----
1971 /**
1972 * Checks if this date is equal to another date.
1973 * <p>
1974 * Compares this {@code LocalDate} with another ensuring that the date is the same.
1975 * <p>
1976 * Only objects of type {@code LocalDate} are compared, other types return false.

```

```

1977     * To compare the dates of two {@code TemporalAccessor} instances, including dates
1978     * in two different chronologies, use {@link ChronoField#EPOCH_DAY} as a comparator.
1979     *
1980     * @param obj the object to check, null returns false
1981     * @return true if this is equal to the other date
1982     */
1983     @Override
1984     public boolean equals(Object obj) {
1985         if (this == obj) {
1986             return true;
1987         }
1988         if (obj instanceof LocalDate) {
1989             return compareTo((LocalDate) obj) == 0;
1990         }
1991         return false;
1992     }
1993
1994     /**
1995     * A hash code for this date.
1996     *
1997     * @return a suitable hash code
1998     */
1999     @Override
2000     public int hashCode() {
2001         int yearValue = year;
2002         int monthValue = month;
2003         int dayValue = day;
2004         return (yearValue & 0xFFFFF800) ^ ((yearValue << 11) + (monthValue << 6) + (dayValue));
2005     }
2006
2007     //-----
2008     /**
2009     * Outputs this date as a {@code String}, such as {@code 2007-12-03}.
2010     * <p>
2011     * The output will be in the ISO-8601 format {@code yyyy-MM-dd}.
2012     *
2013     * @return a string representation of this date, not null
2014     */
2015     @Override
2016     public String toString() {
2017         int yearValue = year;
2018         int monthValue = month;
2019         int dayValue = day;
2020         int absYear = Math.abs(yearValue);
2021         StringBuilder buf = new StringBuilder(10);
2022         if (absYear < 1000) {
2023             if (yearValue < 0) {
2024                 buf.append(yearValue - 10000).deleteCharAt(1);
2025             } else {
2026                 buf.append(yearValue + 10000).deleteCharAt(0);
2027             }
2028         } else {
2029             if (yearValue > 9999) {

```

```

2030         buf.append('+');
2031     }
2032     buf.append(yearValue);
2033 }
2034 return buf.append(monthValue < 10 ? "-0" : "-")
2035     .append(monthValue)
2036     .append(dayValue < 10 ? "-0" : "-")
2037     .append(dayValue)
2038     .toString();
2039 }
2040
2041 //-----
2042 /**
2043  * Writes the object using a
2044  * <a href="../../serialized-form.html#java.time.Ser">dedicated serialized form</a>.
2045  * @serialData
2046  * <pre>
2047  *   out.writeByte(3); // identifies a LocalDate
2048  *   out.writeInt(year);
2049  *   out.writeByte(month);
2050  *   out.writeByte(day);
2051  * </pre>
2052  *
2053  * @return the instance of {@code Ser}, not null
2054  */
2055 private Object writeReplace() {
2056     return new Ser(Ser.LOCAL_DATE_TYPE, this);
2057 }
2058
2059 /**
2060  * Defend against malicious streams.
2061  *
2062  * @param s the stream to read
2063  * @throws InvalidObjectException always
2064  */
2065 private void readObject(ObjectInputStream s) throws InvalidObjectException {
2066     throw new InvalidObjectException("Deserialization via serialization delegate");
2067 }
2068
2069 void writeExternal(DataOutput out) throws IOException {
2070     out.writeInt(year);
2071     out.writeByte(month);
2072     out.writeByte(day);
2073 }
2074
2075 static LocalDate readExternal(DataInput in) throws IOException {
2076     int year = in.readInt();
2077     int month = in.readByte();
2078     int dayOfMonth = in.readByte();
2079     return LocalDate.of(year, month, dayOfMonth);
2080 }
2081
2082 }

```

4.7 LocalDateTime.java

Secuencia 56: java/time/LocalDateTime.java

```
1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
```



```
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time;
63
64  import static java.time.LocalTime.HOURS_PER_DAY;
65  import static java.time.LocalTime.MICROS_PER_DAY;
66  import static java.time.LocalTime.MILLIS_PER_DAY;
67  import static java.time.LocalTime.MINUTES_PER_DAY;
68  import static java.time.LocalTime.NANOS_PER_DAY;
69  import static java.time.LocalTime.NANOS_PER_HOUR;
70  import static java.time.LocalTime.NANOS_PER_MINUTE;
71  import static java.time.LocalTime.NANOS_PER_SECOND;
72  import static java.time.LocalTime.SECONDS_PER_DAY;
73  import static java.time.temporal.ChronoField.NANO_OF_SECOND;
74
75  import java.io.DataInput;
76  import java.io.DataOutput;
77  import java.io.IOException;
78  import java.io.InvalidObjectException;
79  import java.io.ObjectInputStream;
80  import java.io.Serializable;
81  import java.time.chrono.ChronoLocalDateTime;
82  import java.time.format.DateTimeFormatter;
83  import java.time.format.DateTimeParseException;
84  import java.time.temporal.ChronoField;
85  import java.time.temporal.ChronoUnit;
86  import java.time.temporal.Temporal;
87  import java.time.temporal.TemporalAccessor;
88  import java.time.temporal.TemporalAdjuster;
89  import java.time.temporal.TemporalAmount;
90  import java.time.temporal.TemporalField;
91  import java.time.temporal.TemporalQueries;
92  import java.time.temporal.TemporalQuery;
93  import java.time.temporal.TemporalUnit;
94  import java.time.temporal.UnsupportedTemporalTypeException;
95  import java.time.temporal.ValueRange;
96  import java.time.zone.ZoneRules;
97  import java.util.Objects;
98
99  /**
100   * A date-time without a time-zone in the ISO-8601 calendar system,
101   * such as {@code 2007-12-03T10:15:30}.
102   * <p>
103   * {@code LocalDateTime} is an immutable date-time object that represents a date-time,
```

```

104 * often viewed as year-month-day-hour-minute-second. Other date and time fields,
105 * such as day-of-year, day-of-week and week-of-year, can also be accessed.
106 * Time is represented to nanosecond precision.
107 * For example, the value "2nd October 2007 at 13:45.30.123456789" can be
108 * stored in a {@code LocalDateTime}.
109 * <p>
110 * This class does not store or represent a time-zone.
111 * Instead, it is a description of the date, as used for birthdays, combined with
112 * the local time as seen on a wall clock.
113 * It cannot represent an instant on the time-line without additional information
114 * such as an offset or time-zone.
115 * <p>
116 * The ISO-8601 calendar system is the modern civil calendar system used today
117 * in most of the world. It is equivalent to the proleptic Gregorian calendar
118 * system, in which today's rules for leap years are applied for all time.
119 * For most applications written today, the ISO-8601 rules are entirely suitable.
120 * However, any application that makes use of historical dates, and requires them
121 * to be accurate will find the ISO-8601 approach unsuitable.
122 *
123 * <p>
124 * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
125 * class; use of identity-sensitive operations (including reference equality
126 * ({@code ==}), identity hash code, or synchronization) on instances of
127 * {@code LocalDateTime} may have unpredictable results and should be avoided.
128 * The {@code equals} method should be used for comparisons.
129 *
130 * @implSpec
131 * This class is immutable and thread-safe.
132 *
133 * @since 1.8
134 */
135 public final class LocalDateTime
136     implements Temporal, TemporalAdjuster, ChronoLocalDateTime<LocalDate>, Serializable {
137
138     /**
139      * The minimum supported {@code LocalDateTime}, '-999999999-01-01T00:00:00'.
140      * This is the local date-time of midnight at the start of the minimum date.
141      * This combines {@link LocalDate#MIN} and {@link LocalTime#MIN}.
142      * This could be used by an application as a "far past" date-time.
143      */
144     public static final LocalDateTime MIN = LocalDateTime.of(LocalDate.MIN, LocalTime.MIN);
145
146     /**
147      * The maximum supported {@code LocalDateTime}, '+999999999-12-31T23:59:59.999999999'.
148      * This is the local date-time just before midnight at the end of the maximum date.
149      * This combines {@link LocalDate#MAX} and {@link LocalTime#MAX}.
150      * This could be used by an application as a "far future" date-time.
151      */
152     public static final LocalDateTime MAX = LocalDateTime.of(LocalDate.MAX, LocalTime.MAX);
153
154     /**
155      * Serialization version.
156      */
157     private static final long serialVersionUID = 6207766400415563566L;

```

```
157
158 /**
159  * The date part.
160  */
161 private final LocalDate date;
162 /**
163  * The time part.
164  */
165 private final LocalTime time;
166
167 //-----
168 /**
169  * Obtains the current date-time from the system clock in the default time-zone.
170  * <p>
171  * This will query the {@link Clock#systemDefaultZone() system clock} in the default
172  * time-zone to obtain the current date-time.
173  * <p>
174  * Using this method will prevent the ability to use an alternate clock for testing
175  * because the clock is hard-coded.
176  *
177  * @return the current date-time using the system clock and default time-zone, not null
178  */
179 public static LocalDateTime now() {
180     return now(Clock.systemDefaultZone());
181 }
182
183 /**
184  * Obtains the current date-time from the system clock in the specified time-zone.
185  * <p>
186  * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current date-
187  * time.
188  * <p>
189  * Specifying the time-zone avoids dependence on the default time-zone.
190  * <p>
191  * Using this method will prevent the ability to use an alternate clock for testing
192  * because the clock is hard-coded.
193  *
194  * @param zone the zone ID to use, not null
195  * @return the current date-time using the system clock, not null
196  */
197 public static LocalDateTime now(ZoneId zone) {
198     return now(Clock.system(zone));
199 }
200
201 /**
202  * Obtains the current date-time from the specified clock.
203  * <p>
204  * This will query the specified clock to obtain the current date-time.
205  * <p>
206  * Using this method allows the use of an alternate clock for testing.
207  * The alternate clock may be introduced using {@link Clock dependency injection}.
208  *
209  * @param clock the clock to use, not null
210  * @return the current date-time, not null
211  */
```

```

209 public static LocalDateTime now(Clock clock) {
210     Objects.requireNonNull(clock, "clock");
211     final Instant now = clock.instant(); // called once
212     ZoneOffset offset = clock.getZone().getRules().getOffset(now);
213     return ofEpochSecond(now.getEpochSecond(), now.getNano(), offset);
214 }
215
216 //-----
217 /**
218  * Obtains an instance of {@code LocalDateTime} from year, month,
219  * day, hour and minute, setting the second and nanosecond to zero.
220  * <p>
221  * This returns a {@code LocalDateTime} with the specified year, month,
222  * day-of-month, hour and minute.
223  * The day must be valid for the year and month, otherwise an exception will be thrown.
224  * The second and nanosecond fields will be set to zero.
225  *
226  * @param year the year to represent, from MIN_YEAR to MAX_YEAR
227  * @param month the month-of-year to represent, not null
228  * @param dayOfMonth the day-of-month to represent, from 1 to 31
229  * @param hour the hour-of-day to represent, from 0 to 23
230  * @param minute the minute-of-hour to represent, from 0 to 59
231  * @return the local date-time, not null
232  * @throws DateTimeException if the value of any field is out of range,
233  * or if the day-of-month is invalid for the month-year
234  */
235 public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute) {
236     LocalDate date = LocalDate.of(year, month, dayOfMonth);
237     LocalTime time = LocalTime.of(hour, minute);
238     return new LocalDateTime(date, time);
239 }
240
241 /**
242  * Obtains an instance of {@code LocalDateTime} from year, month,
243  * day, hour, minute and second, setting the nanosecond to zero.
244  * <p>
245  * This returns a {@code LocalDateTime} with the specified year, month,
246  * day-of-month, hour, minute and second.
247  * The day must be valid for the year and month, otherwise an exception will be thrown.
248  * The nanosecond field will be set to zero.
249  *
250  * @param year the year to represent, from MIN_YEAR to MAX_YEAR
251  * @param month the month-of-year to represent, not null
252  * @param dayOfMonth the day-of-month to represent, from 1 to 31
253  * @param hour the hour-of-day to represent, from 0 to 23
254  * @param minute the minute-of-hour to represent, from 0 to 59
255  * @param second the second-of-minute to represent, from 0 to 59
256  * @return the local date-time, not null
257  * @throws DateTimeException if the value of any field is out of range,
258  * or if the day-of-month is invalid for the month-year
259  */
260 public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute, int
    second) {

```

```

261     LocalDate date = LocalDate.of(year, month, dayOfMonth);
262     LocalTime time = LocalTime.of(hour, minute, second);
263     return new LocalDateTime(date, time);
264 }
265
266 /**
267  * Obtains an instance of {@code LocalDateTime} from year, month,
268  * day, hour, minute, second and nanosecond.
269  * <p>
270  * This returns a {@code LocalDateTime} with the specified year, month,
271  * day-of-month, hour, minute, second and nanosecond.
272  * The day must be valid for the year and month, otherwise an exception will be thrown.
273  *
274  * @param year the year to represent, from MIN_YEAR to MAX_YEAR
275  * @param month the month-of-year to represent, not null
276  * @param dayOfMonth the day-of-month to represent, from 1 to 31
277  * @param hour the hour-of-day to represent, from 0 to 23
278  * @param minute the minute-of-hour to represent, from 0 to 59
279  * @param second the second-of-minute to represent, from 0 to 59
280  * @param nanoOfSecond the nano-of-second to represent, from 0 to 999,999,999
281  * @return the local date-time, not null
282  * @throws DateTimeException if the value of any field is out of range,
283  * or if the day-of-month is invalid for the month-year
284  */
285 public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute, int
    second, int nanoOfSecond) {
286     LocalDate date = LocalDate.of(year, month, dayOfMonth);
287     LocalTime time = LocalTime.of(hour, minute, second, nanoOfSecond);
288     return new LocalDateTime(date, time);
289 }
290
291 //-----
292 /**
293  * Obtains an instance of {@code LocalDateTime} from year, month,
294  * day, hour and minute, setting the second and nanosecond to zero.
295  * <p>
296  * This returns a {@code LocalDateTime} with the specified year, month,
297  * day-of-month, hour and minute.
298  * The day must be valid for the year and month, otherwise an exception will be thrown.
299  * The second and nanosecond fields will be set to zero.
300  *
301  * @param year the year to represent, from MIN_YEAR to MAX_YEAR
302  * @param month the month-of-year to represent, from 1 (January) to 12 (December)
303  * @param dayOfMonth the day-of-month to represent, from 1 to 31
304  * @param hour the hour-of-day to represent, from 0 to 23
305  * @param minute the minute-of-hour to represent, from 0 to 59
306  * @return the local date-time, not null
307  * @throws DateTimeException if the value of any field is out of range,
308  * or if the day-of-month is invalid for the month-year
309  */
310 public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute) {
311     LocalDate date = LocalDate.of(year, month, dayOfMonth);
312     LocalTime time = LocalTime.of(hour, minute);

```

```

313         return new LocalDateTime(date, time);
314     }
315
316     /**
317      * Obtains an instance of {@code LocalDateTime} from year, month,
318      * day, hour, minute and second, setting the nanosecond to zero.
319      * <p>
320      * This returns a {@code LocalDateTime} with the specified year, month,
321      * day-of-month, hour, minute and second.
322      * The day must be valid for the year and month, otherwise an exception will be thrown.
323      * The nanosecond field will be set to zero.
324      *
325      * @param year    the year to represent, from MIN_YEAR to MAX_YEAR
326      * @param month    the month-of-year to represent, from 1 (January) to 12 (December)
327      * @param dayOfMonth the day-of-month to represent, from 1 to 31
328      * @param hour     the hour-of-day to represent, from 0 to 23
329      * @param minute   the minute-of-hour to represent, from 0 to 59
330      * @param second   the second-of-minute to represent, from 0 to 59
331      * @return the local date-time, not null
332      * @throws DateTimeException if the value of any field is out of range,
333      *         or if the day-of-month is invalid for the month-year
334      */
335     public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int
        second) {
336         LocalDate date = LocalDate.of(year, month, dayOfMonth);
337         LocalTime time = LocalTime.of(hour, minute, second);
338         return new LocalDateTime(date, time);
339     }
340
341     /**
342      * Obtains an instance of {@code LocalDateTime} from year, month,
343      * day, hour, minute, second and nanosecond.
344      * <p>
345      * This returns a {@code LocalDateTime} with the specified year, month,
346      * day-of-month, hour, minute, second and nanosecond.
347      * The day must be valid for the year and month, otherwise an exception will be thrown.
348      *
349      * @param year    the year to represent, from MIN_YEAR to MAX_YEAR
350      * @param month    the month-of-year to represent, from 1 (January) to 12 (December)
351      * @param dayOfMonth the day-of-month to represent, from 1 to 31
352      * @param hour     the hour-of-day to represent, from 0 to 23
353      * @param minute   the minute-of-hour to represent, from 0 to 59
354      * @param second   the second-of-minute to represent, from 0 to 59
355      * @param nanoOfSecond the nano-of-second to represent, from 0 to 999,999,999
356      * @return the local date-time, not null
357      * @throws DateTimeException if the value of any field is out of range,
358      *         or if the day-of-month is invalid for the month-year
359      */
360     public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int
        second, int nanoOfSecond) {
361         LocalDate date = LocalDate.of(year, month, dayOfMonth);
362         LocalTime time = LocalTime.of(hour, minute, second, nanoOfSecond);
363         return new LocalDateTime(date, time);

```

```

364     }
365
366     /**
367      * Obtains an instance of {@code LocalDateTime} from a date and time.
368      *
369      * @param date    the local date, not null
370      * @param time    the local time, not null
371      * @return the local date-time, not null
372      */
373     public static LocalDateTime of(LocalDate date, LocalTime time) {
374         Objects.requireNonNull(date, "date");
375         Objects.requireNonNull(time, "time");
376         return new LocalDateTime(date, time);
377     }
378
379     //-----
380     /**
381      * Obtains an instance of {@code LocalDateTime} from an {@code Instant} and zone ID.
382      * <p>
383      * This creates a local date-time based on the specified instant.
384      * First, the offset from UTC/Greenwich is obtained using the zone ID and instant,
385      * which is simple as there is only one valid offset for each instant.
386      * Then, the instant and offset are used to calculate the local date-time.
387      *
388      * @param instant the instant to create the date-time from, not null
389      * @param zone    the time-zone, which may be an offset, not null
390      * @return the local date-time, not null
391      * @throws DateTimeException if the result exceeds the supported range
392      */
393     public static LocalDateTime ofInstant(Instant instant, ZoneId zone) {
394         Objects.requireNonNull(instant, "instant");
395         Objects.requireNonNull(zone, "zone");
396         ZoneRules rules = zone.getRules();
397         ZoneOffset offset = rules.getOffset(instant);
398         return ofEpochSecond(instant.getEpochSecond(), instant.getNano(), offset);
399     }
400
401     /**
402      * Obtains an instance of {@code LocalDateTime} using seconds from the
403      * epoch of 1970-01-01T00:00:00Z.
404      * <p>
405      * This allows the {@link ChronoField#INSTANT_SECONDS epoch-second} field
406      * to be converted to a local date-time. This is primarily intended for
407      * low-level conversions rather than general application usage.
408      *
409      * @param epochSecond the number of seconds from the epoch of 1970-01-01T00:00:00Z
410      * @param nanoOfSecond the nanosecond within the second, from 0 to 999,999,999
411      * @param offset       the zone offset, not null
412      * @return the local date-time, not null
413      * @throws DateTimeException if the result exceeds the supported range,
414      *         or if the nano-of-second is invalid
415      */
416     public static LocalDateTime ofEpochSecond(long epochSecond, int nanoOfSecond, ZoneOffset offset

```



```

    ) {
417     Objects.requireNonNull(offset, "offset");
418     NANO_OF_SECOND.checkValidValue(nanoOfSecond);
419     long localSecond = epochSecond + offset.getTotalSeconds(); // overflow caught later
420     long localEpochDay = Math.floorDiv(localSecond, SECONDS_PER_DAY);
421     int secsOfDay = (int)Math.floorMod(localSecond, SECONDS_PER_DAY);
422     LocalDate date = LocalDate.ofEpochDay(localEpochDay);
423     LocalTime time = LocalTime.ofNanoOfDay(secsOfDay * NANOS_PER_SECOND + nanoOfSecond);
424     return new LocalDateTime(date, time);
425 }
426
427 //-----
428 /**
429  * Obtains an instance of {@code LocalDateTime} from a temporal object.
430  * <p>
431  * This obtains a local date-time based on the specified temporal.
432  * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
433  * which this factory converts to an instance of {@code LocalDateTime}.
434  * <p>
435  * The conversion extracts and combines the {@code LocalDate} and the
436  * {@code LocalTime} from the temporal object.
437  * Implementations are permitted to perform optimizations such as accessing
438  * those fields that are equivalent to the relevant objects.
439  * <p>
440  * This method matches the signature of the functional interface {@link TemporalQuery}
441  * allowing it to be used as a query via method reference, {@code LocalDateTime::from}.
442  *
443  * @param temporal the temporal object to convert, not null
444  * @return the local date-time, not null
445  * @throws DateTimeException if unable to convert to a {@code LocalDateTime}
446  */
447 public static LocalDateTime from(TemporalAccessor temporal) {
448     if (temporal instanceof LocalDateTime) {
449         return (LocalDateTime) temporal;
450     } else if (temporal instanceof ZonedDateTime) {
451         return ((ZonedDateTime) temporal).toLocalDateTime();
452     } else if (temporal instanceof OffsetDateTime) {
453         return ((OffsetDateTime) temporal).toLocalDateTime();
454     }
455     try {
456         LocalDate date = LocalDate.from(temporal);
457         LocalTime time = LocalTime.from(temporal);
458         return new LocalDateTime(date, time);
459     } catch (DateTimeException ex) {
460         throw new DateTimeException("Unable to obtain LocalDateTime from TemporalAccessor: " +
461             temporal + " of type " + temporal.getClass().getName(), ex);
462     }
463 }
464
465 //-----
466 /**
467  * Obtains an instance of {@code LocalDateTime} from a text string such as {@code 2007-12-03T10
    :15:30}.

```



```

468 * <p>
469 * The string must represent a valid date-time and is parsed using
470 * {@link java.time.format.DateTimeFormatter#ISO_LOCAL_DATE_TIME}.
471 *
472 * @param text the text to parse such as "2007-12-03T10:15:30", not null
473 * @return the parsed local date-time, not null
474 * @throws DateTimeParseException if the text cannot be parsed
475 */
476 public static LocalDateTime parse(CharSequence text) {
477     return parse(text, DateTimeFormatter.ISO_LOCAL_DATE_TIME);
478 }
479
480 /**
481 * Obtains an instance of {@code LocalDateTime} from a text string using a specific formatter.
482 * <p>
483 * The text is parsed using the formatter, returning a date-time.
484 *
485 * @param text the text to parse, not null
486 * @param formatter the formatter to use, not null
487 * @return the parsed local date-time, not null
488 * @throws DateTimeParseException if the text cannot be parsed
489 */
490 public static LocalDateTime parse(CharSequence text, DateTimeFormatter formatter) {
491     Objects.requireNonNull(formatter, "formatter");
492     return formatter.parse(text, LocalDateTime::from);
493 }
494
495 //-----
496 /**
497 * Constructor.
498 *
499 * @param date the date part of the date-time, validated not null
500 * @param time the time part of the date-time, validated not null
501 */
502 private LocalDateTime(LocalDate date, LocalTime time) {
503     this.date = date;
504     this.time = time;
505 }
506
507 /**
508 * Returns a copy of this date-time with the new date and time, checking
509 * to see if a new object is in fact required.
510 *
511 * @param newDate the date of the new date-time, not null
512 * @param newTime the time of the new date-time, not null
513 * @return the date-time, not null
514 */
515 private LocalDateTime with(LocalDate newDate, LocalTime newTime) {
516     if (date == newDate && time == newTime) {
517         return this;
518     }
519     return new LocalDateTime(newDate, newTime);
520 }

```

```

521 //-----
522 /**
523  * Checks if the specified field is supported.
524  * <p>
525  * This checks if this date-time can be queried for the specified field.
526  * If false, then calling the {@link #range(TemporalField) range},
527  * {@link #get(TemporalField) get} and {@link #with(TemporalField, long)}
528  * methods will throw an exception.
529  * <p>
530  * If the field is a {@link ChronoField} then the query is implemented here.
531  * The supported fields are:
532  * <ul>
533  * <li>{@code NANO_OF_SECOND}
534  * <li>{@code NANO_OF_DAY}
535  * <li>{@code MICRO_OF_SECOND}
536  * <li>{@code MICRO_OF_DAY}
537  * <li>{@code MILLI_OF_SECOND}
538  * <li>{@code MILLI_OF_DAY}
539  * <li>{@code SECOND_OF_MINUTE}
540  * <li>{@code SECOND_OF_DAY}
541  * <li>{@code MINUTE_OF_HOUR}
542  * <li>{@code MINUTE_OF_DAY}
543  * <li>{@code HOUR_OF_AMPM}
544  * <li>{@code CLOCK_HOUR_OF_AMPM}
545  * <li>{@code HOUR_OF_DAY}
546  * <li>{@code CLOCK_HOUR_OF_DAY}
547  * <li>{@code AMPM_OF_DAY}
548  * <li>{@code DAY_OF_WEEK}
549  * <li>{@code ALIGNED_DAY_OF_WEEK_IN_MONTH}
550  * <li>{@code ALIGNED_DAY_OF_WEEK_IN_YEAR}
551  * <li>{@code DAY_OF_MONTH}
552  * <li>{@code DAY_OF_YEAR}
553  * <li>{@code EPOCH_DAY}
554  * <li>{@code ALIGNED_WEEK_OF_MONTH}
555  * <li>{@code ALIGNED_WEEK_OF_YEAR}
556  * <li>{@code MONTH_OF_YEAR}
557  * <li>{@code PROLEPTIC_MONTH}
558  * <li>{@code YEAR_OF_ERA}
559  * <li>{@code YEAR}
560  * <li>{@code ERA}
561  * </ul>
562  * All other {@code ChronoField} instances will return false.
563  * <p>
564  * If the field is not a {@code ChronoField}, then the result of this method
565  * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
566  * passing {@code this} as the argument.
567  * Whether the field is supported is determined by the field.
568  *
569  * @param field the field to check, null returns false
570  * @return true if the field is supported on this date-time, false if not
571  */
572 @Override
573

```

```

574 public boolean isSupported(TemporalField field) {
575     if (field instanceof ChronoField) {
576         ChronoField f = (ChronoField) field;
577         return f.isDateBased() || f.isTimeBased();
578     }
579     return field != null && field.isSupportedBy(this);
580 }
581
582 /**
583  * Checks if the specified unit is supported.
584  * <p>
585  * This checks if the specified unit can be added to, or subtracted from, this date-time.
586  * If false, then calling the {@link #plus(long, TemporalUnit)} and
587  * {@link #minus(long, TemporalUnit) minus} methods will throw an exception.
588  * <p>
589  * If the unit is a {@link ChronoUnit} then the query is implemented here.
590  * The supported units are:
591  * <ul>
592  * <li>{@code NANOS}
593  * <li>{@code MICROS}
594  * <li>{@code MILLIS}
595  * <li>{@code SECONDS}
596  * <li>{@code MINUTES}
597  * <li>{@code HOURS}
598  * <li>{@code HALF_DAYS}
599  * <li>{@code DAYS}
600  * <li>{@code WEEKS}
601  * <li>{@code MONTHS}
602  * <li>{@code YEARS}
603  * <li>{@code DECADES}
604  * <li>{@code CENTURIES}
605  * <li>{@code MILLENNIA}
606  * <li>{@code ERAS}
607  * </ul>
608  * All other {@code ChronoUnit} instances will return false.
609  * <p>
610  * If the unit is not a {@code ChronoUnit}, then the result of this method
611  * is obtained by invoking {@code TemporalUnit.isSupportedBy(Temporal)}
612  * passing {@code this} as the argument.
613  * Whether the unit is supported is determined by the unit.
614  *
615  * @param unit the unit to check, null returns false
616  * @return true if the unit can be added/subtracted, false if not
617  */
618 @Override // override for Javadoc
619 public boolean isSupported(TemporalUnit unit) {
620     return ChronoLocalDateTime.super.isSupported(unit);
621 }
622
623 //-----
624 /**
625  * Gets the range of valid values for the specified field.
626  * <p>

```

```

627 * The range object expresses the minimum and maximum valid values for a field.
628 * This date-time is used to enhance the accuracy of the returned range.
629 * If it is not possible to return the range, because the field is not supported
630 * or for some other reason, an exception is thrown.
631 * <p>
632 * If the field is a {@link ChronoField} then the query is implemented here.
633 * The {@link #isSupported(TemporalField) supported fields} will return
634 * appropriate range instances.
635 * All other {@code ChronoField} instances will throw an {@code
        UnsupportedTemporalTypeException}.
636 * <p>
637 * If the field is not a {@code ChronoField}, then the result of this method
638 * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
639 * passing {@code this} as the argument.
640 * Whether the range can be obtained is determined by the field.
641 *
642 * @param field the field to query the range for, not null
643 * @return the range of valid values for the field, not null
644 * @throws DateTimeException if the range for the field cannot be obtained
645 * @throws UnsupportedTemporalTypeException if the field is not supported
646 */
647 @Override
648 public ValueRange range(TemporalField field) {
649     if (field instanceof ChronoField) {
650         ChronoField f = (ChronoField) field;
651         return (f.isTimeBased() ? time.range(field) : date.range(field));
652     }
653     return field.rangeRefinedBy(this);
654 }
655
656 /**
657 * Gets the value of the specified field from this date-time as an {@code int}.
658 * <p>
659 * This queries this date-time for the value of the specified field.
660 * The returned value will always be within the valid range of values for the field.
661 * If it is not possible to return the value, because the field is not supported
662 * or for some other reason, an exception is thrown.
663 * <p>
664 * If the field is a {@link ChronoField} then the query is implemented here.
665 * The {@link #isSupported(TemporalField) supported fields} will return valid
666 * values based on this date-time, except {@code NANO_OF_DAY}, {@code MICRO_OF_DAY},
667 * {@code EPOCH_DAY} and {@code PROLEPTIC_MONTH} which are too large to fit in
668 * an {@code int} and throw a {@code DateTimeException}.
669 * All other {@code ChronoField} instances will throw an {@code
        UnsupportedTemporalTypeException}.
670 * <p>
671 * If the field is not a {@code ChronoField}, then the result of this method
672 * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
673 * passing {@code this} as the argument. Whether the value can be obtained,
674 * and what the value represents, is determined by the field.
675 *
676 * @param field the field to get, not null
677 * @return the value for the field

```

```

678     * @throws DateTimeException if a value for the field cannot be obtained or
679     *       the value is outside the range of valid values for the field
680     * @throws UnsupportedTemporalTypeException if the field is not supported or
681     *       the range of values exceeds an {@code int}
682     * @throws ArithmeticException if numeric overflow occurs
683     */
684     @Override
685     public int get(TemporalField field) {
686         if (field instanceof ChronoField) {
687             ChronoField f = (ChronoField) field;
688             return (f.isTimeBased() ? time.get(field) : date.get(field));
689         }
690         return ChronoLocalDateTime.super.get(field);
691     }
692
693     /**
694     * Gets the value of the specified field from this date-time as a {@code long}.
695     * <p>
696     * This queries this date-time for the value of the specified field.
697     * If it is not possible to return the value, because the field is not supported
698     * or for some other reason, an exception is thrown.
699     * <p>
700     * If the field is a {@link ChronoField} then the query is implemented here.
701     * The {@link #isSupported(TemporalField)} supported fields} will return valid
702     * values based on this date-time.
703     * All other {@code ChronoField} instances will throw an {@code
704         UnsupportedTemporalTypeException}.
705     * <p>
706     * If the field is not a {@code ChronoField}, then the result of this method
707     * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
708     * passing {@code this} as the argument. Whether the value can be obtained,
709     * and what the value represents, is determined by the field.
710     *
711     * @param field the field to get, not null
712     * @return the value for the field
713     * @throws DateTimeException if a value for the field cannot be obtained
714     * @throws UnsupportedTemporalTypeException if the field is not supported
715     * @throws ArithmeticException if numeric overflow occurs
716     */
717     @Override
718     public long getLong(TemporalField field) {
719         if (field instanceof ChronoField) {
720             ChronoField f = (ChronoField) field;
721             return (f.isTimeBased() ? time.getLong(field) : date.getLong(field));
722         }
723         return field.getFrom(this);
724     }
725
726     //-----
727     /**
728     * Gets the {@code LocalDate} part of this date-time.
729     * <p>
730     * This returns a {@code LocalDate} with the same year, month and day

```

```
730     * as this date-time.
731     *
732     * @return the date part of this date-time, not null
733     */
734     @Override
735     public LocalDate toLocalDate() {
736         return date;
737     }
738
739     /**
740     * Gets the year field.
741     * <p>
742     * This method returns the primitive {@code int} value for the year.
743     * <p>
744     * The year returned by this method is proleptic as per {@code get(YEAR)}.
745     * To obtain the year-of-era, use {@code get(YEAR_OF_ERA)}.
746     *
747     * @return the year, from MIN_YEAR to MAX_YEAR
748     */
749     public int getYear() {
750         return date.getYear();
751     }
752
753     /**
754     * Gets the month-of-year field from 1 to 12.
755     * <p>
756     * This method returns the month as an {@code int} from 1 to 12.
757     * Application code is frequently clearer if the enum {@link Month}
758     * is used by calling {@link #getMonth()}.
759     *
760     * @return the month-of-year, from 1 to 12
761     * @see #getMonth()
762     */
763     public int getMonthValue() {
764         return date.getMonthValue();
765     }
766
767     /**
768     * Gets the month-of-year field using the {@code Month} enum.
769     * <p>
770     * This method returns the enum {@link Month} for the month.
771     * This avoids confusion as to what {@code int} values mean.
772     * If you need access to the primitive {@code int} value then the enum
773     * provides the {@link Month#getValue() int value}.
774     *
775     * @return the month-of-year, not null
776     * @see #getMonthValue()
777     */
778     public Month getMonth() {
779         return date.getMonth();
780     }
781
782     /**
```

```

783     * Gets the day-of-month field.
784     * <p>
785     * This method returns the primitive {@code int} value for the day-of-month.
786     *
787     * @return the day-of-month, from 1 to 31
788     */
789     public int getDayOfMonth() {
790         return date.getDayOfMonth();
791     }
792
793     /**
794     * Gets the day-of-year field.
795     * <p>
796     * This method returns the primitive {@code int} value for the day-of-year.
797     *
798     * @return the day-of-year, from 1 to 365, or 366 in a leap year
799     */
800     public int getDayOfYear() {
801         return date.getDayOfYear();
802     }
803
804     /**
805     * Gets the day-of-week field, which is an enum {@code DayOfWeek}.
806     * <p>
807     * This method returns the enum {@link DayOfWeek} for the day-of-week.
808     * This avoids confusion as to what {@code int} values mean.
809     * If you need access to the primitive {@code int} value then the enum
810     * provides the {@link DayOfWeek#getValue() int value}.
811     * <p>
812     * Additional information can be obtained from the {@code DayOfWeek}.
813     * This includes textual names of the values.
814     *
815     * @return the day-of-week, not null
816     */
817     public DayOfWeek getDayOfWeek() {
818         return date.getDayOfWeek();
819     }
820
821     //-----
822     /**
823     * Gets the {@code LocalTime} part of this date-time.
824     * <p>
825     * This returns a {@code LocalTime} with the same hour, minute, second and
826     * nanosecond as this date-time.
827     *
828     * @return the time part of this date-time, not null
829     */
830     @Override
831     public LocalTime toLocalTime() {
832         return time;
833     }
834
835     /**

```

```

836     * Gets the hour-of-day field.
837     *
838     * @return the hour-of-day, from 0 to 23
839     */
840     public int getHour() {
841         return time.getHour();
842     }
843
844     /**
845     * Gets the minute-of-hour field.
846     *
847     * @return the minute-of-hour, from 0 to 59
848     */
849     public int getMinute() {
850         return time.getMinute();
851     }
852
853     /**
854     * Gets the second-of-minute field.
855     *
856     * @return the second-of-minute, from 0 to 59
857     */
858     public int getSecond() {
859         return time.getSecond();
860     }
861
862     /**
863     * Gets the nano-of-second field.
864     *
865     * @return the nano-of-second, from 0 to 999,999,999
866     */
867     public int getNano() {
868         return time.getNano();
869     }
870
871     //-----
872     /**
873     * Returns an adjusted copy of this date-time.
874     * <p>
875     * This returns a {@code LocalDateTime}, based on this one, with the date-time adjusted.
876     * The adjustment takes place using the specified adjuster strategy object.
877     * Read the documentation of the adjuster to understand what adjustment will be made.
878     * <p>
879     * A simple adjuster might simply set the one of the fields, such as the year field.
880     * A more complex adjuster might set the date to the last day of the month.
881     * <p>
882     * A selection of common adjustments is provided in
883     * {@link java.time.temporal.TemporalAdjusters TemporalAdjusters}.
884     * These include finding the "last day of the month" and "next Wednesday".
885     * Key date-time classes also implement the {@code TemporalAdjuster} interface,
886     * such as {@link Month} and {@link java.time.MonthDay MonthDay}.
887     * The adjuster is responsible for handling special cases, such as the varying
888     * lengths of month and leap years.

```



```

889 * <p>
890 * For example this code returns a date on the last day of July:
891 * <pre>
892 * import static java.time.Month.*;
893 * import static java.time.temporal.TemporalAdjusters.*;
894 *
895 * result = localDateTime.with(JULY).with(lastDayOfMonth());
896 * </pre>
897 * <p>
898 * The classes {@link LocalDate} and {@link LocalTime} implement {@code TemporalAdjuster},
899 * thus this method can be used to change the date, time or offset:
900 * <pre>
901 * result = localDateTime.with(date);
902 * result = localDateTime.with(time);
903 * </pre>
904 * <p>
905 * The result of this method is obtained by invoking the
906 * {@link TemporalAdjuster#adjustInto(Temporal)} method on the
907 * specified adjuster passing {@code this} as the argument.
908 * <p>
909 * This instance is immutable and unaffected by this method call.
910 *
911 * @param adjuster the adjuster to use, not null
912 * @return a {@code LocalDateTime} based on {@code this} with the adjustment made, not null
913 * @throws DateTimeException if the adjustment cannot be made
914 * @throws ArithmeticException if numeric overflow occurs
915 */
916 @Override
917 public LocalDateTime with(TemporalAdjuster adjuster) {
918     // optimizations
919     if (adjuster instanceof LocalDate) {
920         return with((LocalDate) adjuster, time);
921     } else if (adjuster instanceof LocalTime) {
922         return with(date, (LocalTime) adjuster);
923     } else if (adjuster instanceof LocalDateTime) {
924         return (LocalDateTime) adjuster;
925     }
926     return (LocalDateTime) adjuster.adjustInto(this);
927 }
928
929 /**
930 * Returns a copy of this date-time with the specified field set to a new value.
931 * <p>
932 * This returns a {@code LocalDateTime}, based on this one, with the value
933 * for the specified field changed.
934 * This can be used to change any supported field, such as the year, month or day-of-month.
935 * If it is not possible to set the value, because the field is not supported or for
936 * some other reason, an exception is thrown.
937 * <p>
938 * In some cases, changing the specified field can cause the resulting date-time to become
939 * invalid,
940 * such as changing the month from 31st January to February would make the day-of-month invalid
941 * .

```

```

940     * In cases like this, the field is responsible for resolving the date. Typically it will
941     choose
942     * <p>
943     * If the field is a {@link ChronoField} then the adjustment is implemented here.
944     * The {@link #isSupported(TemporalField) supported fields} will behave as per
945     * the matching method on {@link LocalDate#with(TemporalField, long) LocalDate}
946     * or {@link LocalTime#with(TemporalField, long) LocalTime}.
947     * All other {@code ChronoField} instances will throw an {@code
948         UnsupportedTemporalTypeException}.
949     * <p>
950     * If the field is not a {@code ChronoField}, then the result of this method
951     * is obtained by invoking {@code TemporalField.adjustInto(Temporal, long)}
952     * passing {@code this} as the argument. In this case, the field determines
953     * whether and how to adjust the instant.
954     * <p>
955     * This instance is immutable and unaffected by this method call.
956     *
957     * @param field the field to set in the result, not null
958     * @param newValue the new value of the field in the result
959     * @return a {@code LocalDateTime} based on {@code this} with the specified field set, not null
960     * @throws DateTimeException if the field cannot be set
961     * @throws UnsupportedTemporalTypeException if the field is not supported
962     * @throws ArithmeticException if numeric overflow occurs
963     */
964     @Override
965     public LocalDateTime with(TemporalField field, long newValue) {
966         if (field instanceof ChronoField) {
967             ChronoField f = (ChronoField) field;
968             if (f.isTimeBased()) {
969                 return with(date, time.with(field, newValue));
970             } else {
971                 return with(date.with(field, newValue), time);
972             }
973         }
974         return field.adjustInto(this, newValue);
975     }
976
977     //-----
978     /**
979     * Returns a copy of this {@code LocalDateTime} with the year altered.
980     * <p>
981     * The time does not affect the calculation and will be the same in the result.
982     * If the day-of-month is invalid for the year, it will be changed to the last valid day of the
983     month.
984     * <p>
985     * This instance is immutable and unaffected by this method call.
986     *
987     * @param year the year to set in the result, from MIN_YEAR to MAX_YEAR
988     * @return a {@code LocalDateTime} based on this date-time with the requested year, not null
989     * @throws DateTimeException if the year value is invalid
990     */
991     public LocalDateTime withYear(int year) {

```

```
990         return with(date.withYear(year), time);
991     }
992
993     /**
994      * Returns a copy of this {@code LocalDateTime} with the month-of-year altered.
995      * <p>
996      * The time does not affect the calculation and will be the same in the result.
997      * If the day-of-month is invalid for the year, it will be changed to the last valid day of the
998      * month.
999      * <p>
1000      * This instance is immutable and unaffected by this method call.
1001      *
1002      * @param month the month-of-year to set in the result, from 1 (January) to 12 (December)
1003      * @return a {@code LocalDateTime} based on this date-time with the requested month, not null
1004      * @throws DateTimeException if the month-of-year value is invalid
1005      */
1006     public LocalDateTime withMonth(int month) {
1007         return with(date.withMonth(month), time);
1008     }
1009
1010     /**
1011      * Returns a copy of this {@code LocalDateTime} with the day-of-month altered.
1012      * <p>
1013      * If the resulting date-time is invalid, an exception is thrown.
1014      * The time does not affect the calculation and will be the same in the result.
1015      * <p>
1016      * This instance is immutable and unaffected by this method call.
1017      *
1018      * @param dayOfMonth the day-of-month to set in the result, from 1 to 28-31
1019      * @return a {@code LocalDateTime} based on this date-time with the requested day, not null
1020      * @throws DateTimeException if the day-of-month value is invalid,
1021      * or if the day-of-month is invalid for the month-year
1022      */
1023     public LocalDateTime withDayOfMonth(int dayOfMonth) {
1024         return with(date.withDayOfMonth(dayOfMonth), time);
1025     }
1026
1027     /**
1028      * Returns a copy of this {@code LocalDateTime} with the day-of-year altered.
1029      * <p>
1030      * If the resulting date-time is invalid, an exception is thrown.
1031      * <p>
1032      * This instance is immutable and unaffected by this method call.
1033      *
1034      * @param dayOfYear the day-of-year to set in the result, from 1 to 365-366
1035      * @return a {@code LocalDateTime} based on this date with the requested day, not null
1036      * @throws DateTimeException if the day-of-year value is invalid,
1037      * or if the day-of-year is invalid for the year
1038      */
1039     public LocalDateTime withDayOfYear(int dayOfYear) {
1040         return with(date.withDayOfYear(dayOfYear), time);
1041     }
```

```
1042 //-----
1043 /**
1044  * Returns a copy of this {@code LocalDateTime} with the hour-of-day altered.
1045  * <p>
1046  * This instance is immutable and unaffected by this method call.
1047  *
1048  * @param hour the hour-of-day to set in the result, from 0 to 23
1049  * @return a {@code LocalDateTime} based on this date-time with the requested hour, not null
1050  * @throws DateTimeException if the hour value is invalid
1051  */
1052 public LocalDateTime withHour(int hour) {
1053     LocalTime newTime = time.withHour(hour);
1054     return with(date, newTime);
1055 }
1056
1057 /**
1058  * Returns a copy of this {@code LocalDateTime} with the minute-of-hour altered.
1059  * <p>
1060  * This instance is immutable and unaffected by this method call.
1061  *
1062  * @param minute the minute-of-hour to set in the result, from 0 to 59
1063  * @return a {@code LocalDateTime} based on this date-time with the requested minute, not null
1064  * @throws DateTimeException if the minute value is invalid
1065  */
1066 public LocalDateTime withMinute(int minute) {
1067     LocalTime newTime = time.withMinute(minute);
1068     return with(date, newTime);
1069 }
1070
1071 /**
1072  * Returns a copy of this {@code LocalDateTime} with the second-of-minute altered.
1073  * <p>
1074  * This instance is immutable and unaffected by this method call.
1075  *
1076  * @param second the second-of-minute to set in the result, from 0 to 59
1077  * @return a {@code LocalDateTime} based on this date-time with the requested second, not null
1078  * @throws DateTimeException if the second value is invalid
1079  */
1080 public LocalDateTime withSecond(int second) {
1081     LocalTime newTime = time.withSecond(second);
1082     return with(date, newTime);
1083 }
1084
1085 /**
1086  * Returns a copy of this {@code LocalDateTime} with the nano-of-second altered.
1087  * <p>
1088  * This instance is immutable and unaffected by this method call.
1089  *
1090  * @param nanoOfSecond the nano-of-second to set in the result, from 0 to 999,999,999
1091  * @return a {@code LocalDateTime} based on this date-time with the requested nanosecond, not
1092         null
1093  * @throws DateTimeException if the nano value is invalid
1094  */
```

```

1094 public LocalDateTime withNano(int nanoOfSecond) {
1095     LocalDateTime newTime = time.withNano(nanoOfSecond);
1096     return with(date, newTime);
1097 }
1098
1099 //-----
1100 /**
1101  * Returns a copy of this {@code LocalDateTime} with the time truncated.
1102  * <p>
1103  * Truncation returns a copy of the original date-time with fields
1104  * smaller than the specified unit set to zero.
1105  * For example, truncating with the {@link ChronoUnit#MINUTES minutes} unit
1106  * will set the second-of-minute and nano-of-second field to zero.
1107  * <p>
1108  * The unit must have a {@link PlainTemporalUnit#getDuration() duration}
1109  * that divides into the length of a standard day without remainder.
1110  * This includes all supplied time units on {@link ChronoUnit} and
1111  * {@link ChronoUnit#DAYS DAYS}. Other units throw an exception.
1112  * <p>
1113  * This instance is immutable and unaffected by this method call.
1114  *
1115  * @param unit the unit to truncate to, not null
1116  * @return a {@code LocalDateTime} based on this date-time with the time truncated, not null
1117  * @throws DateTimeException if unable to truncate
1118  * @throws UnsupportedTemporalTypeException if the unit is not supported
1119  */
1120 public LocalDateTime truncatedTo(TemporalUnit unit) {
1121     return with(date, time.truncatedTo(unit));
1122 }
1123
1124 //-----
1125 /**
1126  * Returns a copy of this date-time with the specified amount added.
1127  * <p>
1128  * This returns a {@code LocalDateTime}, based on this one, with the specified amount added.
1129  * The amount is typically {@link Period} or {@link Duration} but may be
1130  * any other type implementing the {@link TemporalAmount} interface.
1131  * <p>
1132  * The calculation is delegated to the amount object by calling
1133  * {@link TemporalAmount#addTo(Temporal)}. The amount implementation is free
1134  * to implement the addition in any way it wishes, however it typically
1135  * calls back to {@link #plus(long, TemporalUnit)}. Consult the documentation
1136  * of the amount implementation to determine if it can be successfully added.
1137  * <p>
1138  * This instance is immutable and unaffected by this method call.
1139  *
1140  * @param amountToAdd the amount to add, not null
1141  * @return a {@code LocalDateTime} based on this date-time with the addition made, not null
1142  * @throws DateTimeException if the addition cannot be made
1143  * @throws ArithmeticException if numeric overflow occurs
1144  */
1145 @Override
1146 public LocalDateTime plus(TemporalAmount amountToAdd) {

```

```

1147     if (amountToAdd instanceof Period) {
1148         Period periodToAdd = (Period) amountToAdd;
1149         return with(date.plus(periodToAdd), time);
1150     }
1151     Objects.requireNonNull(amountToAdd, "amountToAdd");
1152     return (LocalDateTime) amountToAdd.addTo(this);
1153 }
1154
1155 /**
1156  * Returns a copy of this date-time with the specified amount added.
1157  * <p>
1158  * This returns a {@code LocalDateTime}, based on this one, with the amount
1159  * in terms of the unit added. If it is not possible to add the amount, because the
1160  * unit is not supported or for some other reason, an exception is thrown.
1161  * <p>
1162  * If the field is a {@link ChronoUnit} then the addition is implemented here.
1163  * Date units are added as per {@link LocalDate#plus(long, TemporalUnit)}.
1164  * Time units are added as per {@link LocalTime#plus(long, TemporalUnit)} with
1165  * any overflow in days added equivalent to using {@link #plusDays(long)}.
1166  * <p>
1167  * If the field is not a {@code ChronoUnit}, then the result of this method
1168  * is obtained by invoking {@code TemporalUnit.addTo(Temporal, long)}
1169  * passing {@code this} as the argument. In this case, the unit determines
1170  * whether and how to perform the addition.
1171  * <p>
1172  * This instance is immutable and unaffected by this method call.
1173  *
1174  * @param amountToAdd the amount of the unit to add to the result, may be negative
1175  * @param unit the unit of the amount to add, not null
1176  * @return a {@code LocalDateTime} based on this date-time with the specified amount added, not
1177  *         null
1178  * @throws DateTimeException if the addition cannot be made
1179  * @throws UnsupportedTemporalTypeException if the unit is not supported
1180  * @throws ArithmeticException if numeric overflow occurs
1181  */
1182 @Override
1183 public LocalDateTime plus(long amountToAdd, TemporalUnit unit) {
1184     if (unit instanceof ChronoUnit) {
1185         ChronoUnit f = (ChronoUnit) unit;
1186         switch (f) {
1187             case NANOS: return plusNanos(amountToAdd);
1188             case MICROS: return plusDays(amountToAdd / MICROS_PER_DAY).plusNanos((amountToAdd %
1189                                     MICROS_PER_DAY) * 1000);
1190             case MILLIS: return plusDays(amountToAdd / MILLIS_PER_DAY).plusNanos((amountToAdd %
1191                                     MILLIS_PER_DAY) * 1000_000);
1192             case SECONDS: return plusSeconds(amountToAdd);
1193             case MINUTES: return plusMinutes(amountToAdd);
1194             case HOURS: return plusHours(amountToAdd);
1195             case HALF_DAYS: return plusDays(amountToAdd / 256).plusHours((amountToAdd % 256) *
1196                                     12); // no overflow (256 is multiple of 2)
1197         }
1198     }
1199     return with(date.plus(amountToAdd, unit), time);
1200 }

```

```

1196         return unit.addTo(this, amountToAdd);
1197     }
1198
1199     //-----
1200     /**
1201      * Returns a copy of this {@code LocalDateTime} with the specified number of years added.
1202      * <p>
1203      * This method adds the specified amount to the years field in three steps:
1204      * <ol>
1205      * <li>Add the input years to the year field</li>
1206      * <li>Check if the resulting date would be invalid</li>
1207      * <li>Adjust the day-of-month to the last valid day if necessary</li>
1208      * </ol>
1209      * <p>
1210      * For example, 2008-02-29 (leap year) plus one year would result in the
1211      * invalid date 2009-02-29 (standard year). Instead of returning an invalid
1212      * result, the last valid day of the month, 2009-02-28, is selected instead.
1213      * <p>
1214      * This instance is immutable and unaffected by this method call.
1215      *
1216      * @param years the years to add, may be negative
1217      * @return a {@code LocalDateTime} based on this date-time with the years added, not null
1218      * @throws DateTimeException if the result exceeds the supported date range
1219      */
1220     public LocalDateTime plusYears(long years) {
1221         LocalDate newDate = date.plusYears(years);
1222         return with(newDate, time);
1223     }
1224
1225     /**
1226      * Returns a copy of this {@code LocalDateTime} with the specified number of months added.
1227      * <p>
1228      * This method adds the specified amount to the months field in three steps:
1229      * <ol>
1230      * <li>Add the input months to the month-of-year field</li>
1231      * <li>Check if the resulting date would be invalid</li>
1232      * <li>Adjust the day-of-month to the last valid day if necessary</li>
1233      * </ol>
1234      * <p>
1235      * For example, 2007-03-31 plus one month would result in the invalid date
1236      * 2007-04-31. Instead of returning an invalid result, the last valid day
1237      * of the month, 2007-04-30, is selected instead.
1238      * <p>
1239      * This instance is immutable and unaffected by this method call.
1240      *
1241      * @param months the months to add, may be negative
1242      * @return a {@code LocalDateTime} based on this date-time with the months added, not null
1243      * @throws DateTimeException if the result exceeds the supported date range
1244      */
1245     public LocalDateTime plusMonths(long months) {
1246         LocalDate newDate = date.plusMonths(months);
1247         return with(newDate, time);
1248     }

```



```

1249
1250 /**
1251  * Returns a copy of this {@code LocalDateTime} with the specified number of weeks added.
1252  * <p>
1253  * This method adds the specified amount in weeks to the days field incrementing
1254  * the month and year fields as necessary to ensure the result remains valid.
1255  * The result is only invalid if the maximum/minimum year is exceeded.
1256  * <p>
1257  * For example, 2008-12-31 plus one week would result in 2009-01-07.
1258  * <p>
1259  * This instance is immutable and unaffected by this method call.
1260  *
1261  * @param weeks the weeks to add, may be negative
1262  * @return a {@code LocalDateTime} based on this date-time with the weeks added, not null
1263  * @throws DateTimeException if the result exceeds the supported date range
1264  */
1265 public LocalDateTime plusWeeks(long weeks) {
1266     LocalDate newDate = date.plusWeeks(weeks);
1267     return with(newDate, time);
1268 }
1269
1270 /**
1271  * Returns a copy of this {@code LocalDateTime} with the specified number of days added.
1272  * <p>
1273  * This method adds the specified amount to the days field incrementing the
1274  * month and year fields as necessary to ensure the result remains valid.
1275  * The result is only invalid if the maximum/minimum year is exceeded.
1276  * <p>
1277  * For example, 2008-12-31 plus one day would result in 2009-01-01.
1278  * <p>
1279  * This instance is immutable and unaffected by this method call.
1280  *
1281  * @param days the days to add, may be negative
1282  * @return a {@code LocalDateTime} based on this date-time with the days added, not null
1283  * @throws DateTimeException if the result exceeds the supported date range
1284  */
1285 public LocalDateTime plusDays(long days) {
1286     LocalDate newDate = date.plusDays(days);
1287     return with(newDate, time);
1288 }
1289
1290 //-----
1291 /**
1292  * Returns a copy of this {@code LocalDateTime} with the specified number of hours added.
1293  * <p>
1294  * This instance is immutable and unaffected by this method call.
1295  *
1296  * @param hours the hours to add, may be negative
1297  * @return a {@code LocalDateTime} based on this date-time with the hours added, not null
1298  * @throws DateTimeException if the result exceeds the supported date range
1299  */
1300 public LocalDateTime plusHours(long hours) {
1301     return plusWithOverflow(date, hours, 0, 0, 0, 1);

```



```

1302     }
1303
1304     /**
1305      * Returns a copy of this {@code LocalDateTime} with the specified number of minutes added.
1306      * <p>
1307      * This instance is immutable and unaffected by this method call.
1308      *
1309      * @param minutes the minutes to add, may be negative
1310      * @return a {@code LocalDateTime} based on this date-time with the minutes added, not null
1311      * @throws DateTimeException if the result exceeds the supported date range
1312      */
1313     public LocalDateTime plusMinutes(long minutes) {
1314         return plusWithOverflow(date, 0, minutes, 0, 0, 1);
1315     }
1316
1317     /**
1318      * Returns a copy of this {@code LocalDateTime} with the specified number of seconds added.
1319      * <p>
1320      * This instance is immutable and unaffected by this method call.
1321      *
1322      * @param seconds the seconds to add, may be negative
1323      * @return a {@code LocalDateTime} based on this date-time with the seconds added, not null
1324      * @throws DateTimeException if the result exceeds the supported date range
1325      */
1326     public LocalDateTime plusSeconds(long seconds) {
1327         return plusWithOverflow(date, 0, 0, seconds, 0, 1);
1328     }
1329
1330     /**
1331      * Returns a copy of this {@code LocalDateTime} with the specified number of nanoseconds added.
1332      * <p>
1333      * This instance is immutable and unaffected by this method call.
1334      *
1335      * @param nanos the nanos to add, may be negative
1336      * @return a {@code LocalDateTime} based on this date-time with the nanoseconds added, not null
1337      * @throws DateTimeException if the result exceeds the supported date range
1338      */
1339     public LocalDateTime plusNanos(long nanos) {
1340         return plusWithOverflow(date, 0, 0, 0, nanos, 1);
1341     }
1342
1343     //-----
1344     /**
1345      * Returns a copy of this date-time with the specified amount subtracted.
1346      * <p>
1347      * This returns a {@code LocalDateTime}, based on this one, with the specified amount
1348      * subtracted.
1349      *
1350      * The amount is typically {@link Period} or {@link Duration} but may be
1351      * any other type implementing the {@link TemporalAmount} interface.
1352      * <p>
1353      * The calculation is delegated to the amount object by calling
1354      * {@link TemporalAmount#subtractFrom(Temporal)}. The amount implementation is free
1355      * to implement the subtraction in any way it wishes, however it typically

```

```

1354 * calls back to {@link #minus(long, TemporalUnit)}. Consult the documentation
1355 * of the amount implementation to determine if it can be successfully subtracted.
1356 * <p>
1357 * This instance is immutable and unaffected by this method call.
1358 *
1359 * @param amountToSubtract the amount to subtract, not null
1360 * @return a {@code LocalDateTime} based on this date-time with the subtraction made, not null
1361 * @throws DateTimeException if the subtraction cannot be made
1362 * @throws ArithmeticException if numeric overflow occurs
1363 */
1364 @Override
1365 public LocalDateTime minus(TemporalAmount amountToSubtract) {
1366     if (amountToSubtract instanceof Period) {
1367         Period periodToSubtract = (Period) amountToSubtract;
1368         return with(date.minus(periodToSubtract), time);
1369     }
1370     Objects.requireNonNull(amountToSubtract, "amountToSubtract");
1371     return (LocalDateTime) amountToSubtract.subtractFrom(this);
1372 }
1373
1374 /**
1375 * Returns a copy of this date-time with the specified amount subtracted.
1376 * <p>
1377 * This returns a {@code LocalDateTime}, based on this one, with the amount
1378 * in terms of the unit subtracted. If it is not possible to subtract the amount,
1379 * because the unit is not supported or for some other reason, an exception is thrown.
1380 * <p>
1381 * This method is equivalent to {@link #plus(long, TemporalUnit)} with the amount negated.
1382 * See that method for a full description of how addition, and thus subtraction, works.
1383 * <p>
1384 * This instance is immutable and unaffected by this method call.
1385 *
1386 * @param amountToSubtract the amount of the unit to subtract from the result, may be negative
1387 * @param unit the unit of the amount to subtract, not null
1388 * @return a {@code LocalDateTime} based on this date-time with the specified amount subtracted
1389 *         , not null
1390 * @throws DateTimeException if the subtraction cannot be made
1391 * @throws UnsupportedTemporalTypeException if the unit is not supported
1392 * @throws ArithmeticException if numeric overflow occurs
1393 */
1394 @Override
1395 public LocalDateTime minus(long amountToSubtract, TemporalUnit unit) {
1396     return (amountToSubtract == Long.MIN_VALUE ? plus(Long.MAX_VALUE, unit).plus(1, unit) :
1397         plus(-amountToSubtract, unit));
1398 }
1399
1400 //-----
1401 /**
1402 * Returns a copy of this {@code LocalDateTime} with the specified number of years subtracted.
1403 * <p>
1404 * This method subtracts the specified amount from the years field in three steps:
1405 * <ol>
1406 * <li>Subtract the input years from the year field</li>

```

```

1405     * <li>Check if the resulting date would be invalid</li>
1406     * <li>Adjust the day-of-month to the last valid day if necessary</li>
1407     * </ol>
1408     * <p>
1409     * For example, 2008-02-29 (leap year) minus one year would result in the
1410     * invalid date 2009-02-29 (standard year). Instead of returning an invalid
1411     * result, the last valid day of the month, 2009-02-28, is selected instead.
1412     * <p>
1413     * This instance is immutable and unaffected by this method call.
1414     *
1415     * @param years the years to subtract, may be negative
1416     * @return a {@code LocalDateTime} based on this date-time with the years subtracted, not null
1417     * @throws DateTimeException if the result exceeds the supported date range
1418     */
1419     public LocalDateTime minusYears(long years) {
1420         return (years == Long.MIN_VALUE ? plusYears(Long.MAX_VALUE).plusYears(1) : plusYears(-years
1421         ));
1422     }
1423     /**
1424     * Returns a copy of this {@code LocalDateTime} with the specified number of months subtracted.
1425     * <p>
1426     * This method subtracts the specified amount from the months field in three steps:
1427     * <ol>
1428     * <li>Subtract the input months from the month-of-year field</li>
1429     * <li>Check if the resulting date would be invalid</li>
1430     * <li>Adjust the day-of-month to the last valid day if necessary</li>
1431     * </ol>
1432     * <p>
1433     * For example, 2007-03-31 minus one month would result in the invalid date
1434     * 2007-04-31. Instead of returning an invalid result, the last valid day
1435     * of the month, 2007-04-30, is selected instead.
1436     * <p>
1437     * This instance is immutable and unaffected by this method call.
1438     *
1439     * @param months the months to subtract, may be negative
1440     * @return a {@code LocalDateTime} based on this date-time with the months subtracted, not null
1441     * @throws DateTimeException if the result exceeds the supported date range
1442     */
1443     public LocalDateTime minusMonths(long months) {
1444         return (months == Long.MIN_VALUE ? plusMonths(Long.MAX_VALUE).plusMonths(1) : plusMonths(-
1445         months));
1446     }
1447     /**
1448     * Returns a copy of this {@code LocalDateTime} with the specified number of weeks subtracted.
1449     * <p>
1450     * This method subtracts the specified amount in weeks from the days field decrementing
1451     * the month and year fields as necessary to ensure the result remains valid.
1452     * The result is only invalid if the maximum/minimum year is exceeded.
1453     * <p>
1454     * For example, 2009-01-07 minus one week would result in 2008-12-31.
1455     * <p>

```

```

1456     * This instance is immutable and unaffected by this method call.
1457     *
1458     * @param weeks the weeks to subtract, may be negative
1459     * @return a {@code LocalDateTime} based on this date-time with the weeks subtracted, not null
1460     * @throws DateTimeException if the result exceeds the supported date range
1461     */
1462     public LocalDateTime minusWeeks(long weeks) {
1463         return (weeks == Long.MIN_VALUE ? plusWeeks(Long.MAX_VALUE).plusWeeks(1) : plusWeeks(-weeks
1464         ));
1465     }
1466     /**
1467     * Returns a copy of this {@code LocalDateTime} with the specified number of days subtracted.
1468     * <p>
1469     * This method subtracts the specified amount from the days field decrementing the
1470     * month and year fields as necessary to ensure the result remains valid.
1471     * The result is only invalid if the maximum/minimum year is exceeded.
1472     * <p>
1473     * For example, 2009-01-01 minus one day would result in 2008-12-31.
1474     * <p>
1475     * This instance is immutable and unaffected by this method call.
1476     *
1477     * @param days the days to subtract, may be negative
1478     * @return a {@code LocalDateTime} based on this date-time with the days subtracted, not null
1479     * @throws DateTimeException if the result exceeds the supported date range
1480     */
1481     public LocalDateTime minusDays(long days) {
1482         return (days == Long.MIN_VALUE ? plusDays(Long.MAX_VALUE).plusDays(1) : plusDays(-days));
1483     }
1484
1485     //-----
1486     /**
1487     * Returns a copy of this {@code LocalDateTime} with the specified number of hours subtracted.
1488     * <p>
1489     * This instance is immutable and unaffected by this method call.
1490     *
1491     * @param hours the hours to subtract, may be negative
1492     * @return a {@code LocalDateTime} based on this date-time with the hours subtracted, not null
1493     * @throws DateTimeException if the result exceeds the supported date range
1494     */
1495     public LocalDateTime minusHours(long hours) {
1496         return plusWithOverflow(date, hours, 0, 0, 0, -1);
1497     }
1498
1499     /**
1500     * Returns a copy of this {@code LocalDateTime} with the specified number of minutes subtracted
1501     * .
1502     * <p>
1503     * This instance is immutable and unaffected by this method call.
1504     *
1505     * @param minutes the minutes to subtract, may be negative
1506     * @return a {@code LocalDateTime} based on this date-time with the minutes subtracted, not
1507     * null

```

```

1506     * @throws DateTimeException if the result exceeds the supported date range
1507     */
1508     public LocalDateTime minusMinutes(long minutes) {
1509         return plusWithOverflow(date, 0, minutes, 0, 0, -1);
1510     }
1511
1512     /**
1513     * Returns a copy of this {@code LocalDateTime} with the specified number of seconds subtracted
1514     *
1515     * <p>
1516     * This instance is immutable and unaffected by this method call.
1517     *
1518     * @param seconds the seconds to subtract, may be negative
1519     * @return a {@code LocalDateTime} based on this date-time with the seconds subtracted, not
1520     *         null
1521     * @throws DateTimeException if the result exceeds the supported date range
1522     */
1523     public LocalDateTime minusSeconds(long seconds) {
1524         return plusWithOverflow(date, 0, 0, seconds, 0, -1);
1525     }
1526
1527     /**
1528     * Returns a copy of this {@code LocalDateTime} with the specified number of nanoseconds
1529     * subtracted.
1530     *
1531     * <p>
1532     * This instance is immutable and unaffected by this method call.
1533     *
1534     * @param nanos the nanos to subtract, may be negative
1535     * @return a {@code LocalDateTime} based on this date-time with the nanoseconds subtracted, not
1536     *         null
1537     * @throws DateTimeException if the result exceeds the supported date range
1538     */
1539     public LocalDateTime minusNanos(long nanos) {
1540         return plusWithOverflow(date, 0, 0, 0, nanos, -1);
1541     }
1542
1543     //-----
1544     /**
1545     * Returns a copy of this {@code LocalDateTime} with the specified period added.
1546     *
1547     * <p>
1548     * This instance is immutable and unaffected by this method call.
1549     *
1550     * @param newDate the new date to base the calculation on, not null
1551     * @param hours the hours to add, may be negative
1552     * @param minutes the minutes to add, may be negative
1553     * @param seconds the seconds to add, may be negative
1554     * @param nanos the nanos to add, may be negative
1555     * @param sign the sign to determine add or subtract
1556     * @return the combined result, not null
1557     */
1558     private LocalDateTime plusWithOverflow(LocalDate newDate, long hours, long minutes, long
1559         seconds, long nanos, int sign) {
1560         // 9223372036854775808 long, 2147483648 int

```

```

1554     if ((hours | minutes | seconds | nanos) == 0) {
1555         return with(newDate, time);
1556     }
1557     long totDays = nanos / NANOS_PER_DAY +           // max/24*60*60*1B
1558         seconds / SECONDS_PER_DAY +                 // max/24*60*60
1559         minutes / MINUTES_PER_DAY +                 // max/24*60
1560         hours / HOURS_PER_DAY;                      // max/24
1561     totDays *= sign;                                 // total max*0.4237...
1562     long totNanos = nanos % NANOS_PER_DAY +          // max 864000000000000
1563         (seconds % SECONDS_PER_DAY) * NANOS_PER_SECOND + // max 864000000000000
1564         (minutes % MINUTES_PER_DAY) * NANOS_PER_MINUTE + // max 864000000000000
1565         (hours % HOURS_PER_DAY) * NANOS_PER_HOUR;     // max 864000000000000
1566     long curNoD = time.toNanoOfDay();                // max 864000000000000
1567     totNanos = totNanos * sign + curNoD;             // total 432000000000000
1568     totDays += Math.floorDiv(totNanos, NANOS_PER_DAY);
1569     long newNoD = Math.floorMod(totNanos, NANOS_PER_DAY);
1570     LocalTime newTime = (newNoD == curNoD ? time : LocalTime.ofNanoOfDay(newNoD));
1571     return with(newDate.plusDays(totDays), newTime);
1572 }
1573
1574 //-----
1575 /**
1576  * Queries this date-time using the specified query.
1577  * <p>
1578  * This queries this date-time using the specified query strategy object.
1579  * The {@code TemporalQuery} object defines the logic to be used to
1580  * obtain the result. Read the documentation of the query to understand
1581  * what the result of this method will be.
1582  * <p>
1583  * The result of this method is obtained by invoking the
1584  * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
1585  * specified query passing {@code this} as the argument.
1586  *
1587  * @param <R> the type of the result
1588  * @param query the query to invoke, not null
1589  * @return the query result, null may be returned (defined by the query)
1590  * @throws DateTimeException if unable to query (defined by the query)
1591  * @throws ArithmeticException if numeric overflow occurs (defined by the query)
1592  */
1593 @SuppressWarnings("unchecked")
1594 @Override // override for Javadoc
1595 public <R> R query(TemporalQuery<R> query) {
1596     if (query == TemporalQueries.localDate()) {
1597         return (R) date;
1598     }
1599     return ChronoLocalDateTime.super.query(query);
1600 }
1601
1602 /**
1603  * Adjusts the specified temporal object to have the same date and time as this object.
1604  * <p>
1605  * This returns a temporal object of the same observable type as the input
1606  * with the date and time changed to be the same as this.

```

```

1607 * <p>
1608 * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
1609 * twice, passing {@link ChronoField#EPOCH_DAY} and
1610 * {@link ChronoField#NANO_OF_DAY} as the fields.
1611 * <p>
1612 * In most cases, it is clearer to reverse the calling pattern by using
1613 * {@link Temporal#with(TemporalAdjuster)}:
1614 * <pre>
1615 *     // these two lines are equivalent, but the second approach is recommended
1616 *     temporal = thisLocalDateTime.adjustInto(temporal);
1617 *     temporal = temporal.with(thisLocalDateTime);
1618 * </pre>
1619 * <p>
1620 * This instance is immutable and unaffected by this method call.
1621 *
1622 * @param temporal the target object to be adjusted, not null
1623 * @return the adjusted object, not null
1624 * @throws DateTimeException if unable to make the adjustment
1625 * @throws ArithmeticException if numeric overflow occurs
1626 */
1627 @Override // override for Javadoc
1628 public Temporal adjustInto(Temporal temporal) {
1629     return ChronoLocalDateTime.super.adjustInto(temporal);
1630 }
1631
1632 /**
1633 * Calculates the amount of time until another date-time in terms of the specified unit.
1634 * <p>
1635 * This calculates the amount of time between two {@code LocalDateTime}
1636 * objects in terms of a single {@code TemporalUnit}.
1637 * The start and end points are {@code this} and the specified date-time.
1638 * The result will be negative if the end is before the start.
1639 * The {@code Temporal} passed to this method is converted to a
1640 * {@code LocalDateTime} using {@link #from(TemporalAccessor)}.
1641 * For example, the amount in days between two date-times can be calculated
1642 * using {@code startDateTime.until(endDateTime, DAYS)}.
1643 * <p>
1644 * The calculation returns a whole number, representing the number of
1645 * complete units between the two date-times.
1646 * For example, the amount in months between 2012-06-15T00:00 and 2012-08-14T23:59
1647 * will only be one month as it is one minute short of two months.
1648 * <p>
1649 * There are two equivalent ways of using this method.
1650 * The first is to invoke this method.
1651 * The second is to use {@link TemporalUnit#between(Temporal, Temporal)}:
1652 * <pre>
1653 *     // these two lines are equivalent
1654 *     amount = start.until(end, MONTHS);
1655 *     amount = MONTHS.between(start, end);
1656 * </pre>
1657 * The choice should be made based on which makes the code more readable.
1658 * <p>
1659 * The calculation is implemented in this method for {@link ChronoUnit}.

```



```

1660 * The units {@code NANOS}, {@code MICROS}, {@code MILLIS}, {@code SECONDS},
1661 * {@code MINUTES}, {@code HOURS} and {@code HALF_DAYS}, {@code DAYS},
1662 * {@code WEEKS}, {@code MONTHS}, {@code YEARS}, {@code DECADES},
1663 * {@code CENTURIES}, {@code MILLENNIA} and {@code ERAS} are supported.
1664 * Other {@code ChronoUnit} values will throw an exception.
1665 * <p>
1666 * If the unit is not a {@code ChronoUnit}, then the result of this method
1667 * is obtained by invoking {@code TemporalUnit.between(Temporal, Temporal)}
1668 * passing {@code this} as the first argument and the converted input temporal
1669 * as the second argument.
1670 * <p>
1671 * This instance is immutable and unaffected by this method call.
1672 *
1673 * @param endExclusive the end date, exclusive, which is converted to a {@code LocalDateTime},
    not null
1674 * @param unit the unit to measure the amount in, not null
1675 * @return the amount of time between this date-time and the end date-time
1676 * @throws DateTimeException if the amount cannot be calculated, or the end
1677 * temporal cannot be converted to a {@code LocalDateTime}
1678 * @throws UnsupportedTemporalTypeException if the unit is not supported
1679 * @throws ArithmeticException if numeric overflow occurs
1680 */
1681 @Override
1682 public long until(Temporal endExclusive, TemporalUnit unit) {
1683     LocalDateTime end = LocalDateTime.from(endExclusive);
1684     if (unit instanceof ChronoUnit) {
1685         if (unit.isTimeBased()) {
1686             long amount = date.daysUntil(end.date);
1687             if (amount == 0) {
1688                 return time.until(end.time, unit);
1689             }
1690             long timePart = end.time.toNanoOfDay() - time.toNanoOfDay();
1691             if (amount > 0) {
1692                 amount--; // safe
1693                 timePart += NANOS_PER_DAY; // safe
1694             } else {
1695                 amount++; // safe
1696                 timePart -= NANOS_PER_DAY; // safe
1697             }
1698             switch ((ChronoUnit) unit) {
1699                 case NANOS:
1700                     amount = Math.multiplyExact(amount, NANOS_PER_DAY);
1701                     break;
1702                 case MICROS:
1703                     amount = Math.multiplyExact(amount, MICROS_PER_DAY);
1704                     timePart = timePart / 1000;
1705                     break;
1706                 case MILLIS:
1707                     amount = Math.multiplyExact(amount, MILLIS_PER_DAY);
1708                     timePart = timePart / 1_000_000;
1709                     break;
1710                 case SECONDS:
1711                     amount = Math.multiplyExact(amount, SECONDS_PER_DAY);

```



```

1712         timePart = timePart / NANOS_PER_SECOND;
1713         break;
1714     case MINUTES:
1715         amount = Math.multiplyExact(amount, MINUTES_PER_DAY);
1716         timePart = timePart / NANOS_PER_MINUTE;
1717         break;
1718     case HOURS:
1719         amount = Math.multiplyExact(amount, HOURS_PER_DAY);
1720         timePart = timePart / NANOS_PER_HOUR;
1721         break;
1722     case HALF_DAYS:
1723         amount = Math.multiplyExact(amount, 2);
1724         timePart = timePart / (NANOS_PER_HOUR * 12);
1725         break;
1726     }
1727     return Math.addExact(amount, timePart);
1728 }
1729 LocalDate endDate = end.date;
1730 if (endDate.isAfter(date) && end.time.isBefore(time)) {
1731     endDate = endDate.minusDays(1);
1732 } else if (endDate.isBefore(date) && end.time.isAfter(time)) {
1733     endDate = endDate.plusDays(1);
1734 }
1735 return date.until(endDate, unit);
1736 }
1737 return unit.between(this, end);
1738 }
1739
1740 /**
1741  * Formats this date-time using the specified formatter.
1742  * <p>
1743  * This date-time will be passed to the formatter to produce a string.
1744  *
1745  * @param formatter the formatter to use, not null
1746  * @return the formatted date-time string, not null
1747  * @throws DateTimeException if an error occurs during printing
1748  */
1749 @Override // override for Javadoc and performance
1750 public String format(DateTimeFormatter formatter) {
1751     Objects.requireNonNull(formatter, "formatter");
1752     return formatter.format(this);
1753 }
1754
1755 //-----
1756 /**
1757  * Combines this date-time with an offset to create an {@code OffsetDateTime}.
1758  * <p>
1759  * This returns an {@code OffsetDateTime} formed from this date-time at the specified offset.
1760  * All possible combinations of date-time and offset are valid.
1761  *
1762  * @param offset the offset to combine with, not null
1763  * @return the offset date-time formed from this date-time and the specified offset, not null
1764  */

```

```

1765 public OffsetDateTime atOffset(ZoneOffset offset) {
1766     return OffsetDateTime.of(this, offset);
1767 }
1768
1769 /**
1770  * Combines this date-time with a time-zone to create a {@code ZonedDateTime}.
1771  * <p>
1772  * This returns a {@code ZonedDateTime} formed from this date-time at the
1773  * specified time-zone. The result will match this date-time as closely as possible.
1774  * Time-zone rules, such as daylight savings, mean that not every local date-time
1775  * is valid for the specified zone, thus the local date-time may be adjusted.
1776  * <p>
1777  * The local date-time is resolved to a single instant on the time-line.
1778  * This is achieved by finding a valid offset from UTC/Greenwich for the local
1779  * date-time as defined by the {@link ZoneRules rules} of the zone ID.
1780  * <p>
1781  * In most cases, there is only one valid offset for a local date-time.
1782  * In the case of an overlap, where clocks are set back, there are two valid offsets.
1783  * This method uses the earlier offset typically corresponding to "summer".
1784  * <p>
1785  * In the case of a gap, where clocks jump forward, there is no valid offset.
1786  * Instead, the local date-time is adjusted to be later by the length of the gap.
1787  * For a typical one hour daylight savings change, the local date-time will be
1788  * moved one hour later into the offset typically corresponding to "summer".
1789  * <p>
1790  * To obtain the later offset during an overlap, call
1791  * {@link ZonedDateTime#withLaterOffsetAtOverlap()} on the result of this method.
1792  * To throw an exception when there is a gap or overlap, use
1793  * {@link ZonedDateTime#ofStrict(LocalDateTime, ZoneOffset, ZoneId)}.
1794  *
1795  * @param zone the time-zone to use, not null
1796  * @return the zoned date-time formed from this date-time, not null
1797  */
1798 @Override
1799 public ZonedDateTime atZone(ZoneId zone) {
1800     return ZonedDateTime.of(this, zone);
1801 }
1802
1803 //-----
1804 /**
1805  * Compares this date-time to another date-time.
1806  * <p>
1807  * The comparison is primarily based on the date-time, from earliest to latest.
1808  * It is "consistent with equals", as defined by {@link Comparable}.
1809  * <p>
1810  * If all the date-times being compared are instances of {@code LocalDateTime},
1811  * then the comparison will be entirely based on the date-time.
1812  * If some dates being compared are in different chronologies, then the
1813  * chronology is also considered, see {@link ChronoLocalDateTime#compareTo}.
1814  *
1815  * @param other the other date-time to compare to, not null
1816  * @return the comparator value, negative if less, positive if greater
1817  */

```

```

1818 @Override // override for Javadoc and performance
1819 public int compareTo(ChronoLocalDateTime<?> other) {
1820     if (other instanceof LocalDateTime) {
1821         return compareTo((LocalDateTime) other);
1822     }
1823     return ChronoLocalDateTime.super.compareTo(other);
1824 }
1825
1826 private int compareTo(LocalDateTime other) {
1827     int cmp = date.compareTo0(other.toLocalDate());
1828     if (cmp == 0) {
1829         cmp = time.compareTo(other.toLocalTime());
1830     }
1831     return cmp;
1832 }
1833
1834 /**
1835  * Checks if this date-time is after the specified date-time.
1836  * <p>
1837  * This checks to see if this date-time represents a point on the
1838  * local time-line after the other date-time.
1839  * <pre>
1840  *   LocalDate a = LocalDateTime.of(2012, 6, 30, 12, 00);
1841  *   LocalDate b = LocalDateTime.of(2012, 7, 1, 12, 00);
1842  *   a.isAfter(b) == false
1843  *   a.isAfter(a) == false
1844  *   b.isAfter(a) == true
1845  * </pre>
1846  * <p>
1847  * This method only considers the position of the two date-times on the local time-line.
1848  * It does not take into account the chronology, or calendar system.
1849  * This is different from the comparison in {@link #compareTo(ChronoLocalDateTime)},
1850  * but is the same approach as {@link ChronoLocalDateTime#timeLineOrder()}.
1851  *
1852  * @param other the other date-time to compare to, not null
1853  * @return true if this date-time is after the specified date-time
1854  */
1855 @Override // override for Javadoc and performance
1856 public boolean isAfter(ChronoLocalDateTime<?> other) {
1857     if (other instanceof LocalDateTime) {
1858         return compareTo((LocalDateTime) other) > 0;
1859     }
1860     return ChronoLocalDateTime.super.isAfter(other);
1861 }
1862
1863 /**
1864  * Checks if this date-time is before the specified date-time.
1865  * <p>
1866  * This checks to see if this date-time represents a point on the
1867  * local time-line before the other date-time.
1868  * <pre>
1869  *   LocalDate a = LocalDateTime.of(2012, 6, 30, 12, 00);
1870  *   LocalDate b = LocalDateTime.of(2012, 7, 1, 12, 00);

```

```

1871 * a.isBefore(b) == true
1872 * a.isBefore(a) == false
1873 * b.isBefore(a) == false
1874 * </pre>
1875 * <p>
1876 * This method only considers the position of the two date-times on the local time-line.
1877 * It does not take into account the chronology, or calendar system.
1878 * This is different from the comparison in {@link #compareTo(ChronoLocalDateTime)},
1879 * but is the same approach as {@link ChronoLocalDateTime#timeLineOrder()}.
1880 *
1881 * @param other the other date-time to compare to, not null
1882 * @return true if this date-time is before the specified date-time
1883 */
1884 @Override // override for Javadoc and performance
1885 public boolean isBefore(ChronoLocalDateTime<?> other) {
1886     if (other instanceof LocalDateTime) {
1887         return compareTo((LocalDateTime) other) < 0;
1888     }
1889     return ChronoLocalDateTime.super.isBefore(other);
1890 }
1891
1892 /**
1893 * Checks if this date-time is equal to the specified date-time.
1894 * <p>
1895 * This checks to see if this date-time represents the same point on the
1896 * local time-line as the other date-time.
1897 * <pre>
1898 * LocalDate a = LocalDateTime.of(2012, 6, 30, 12, 00);
1899 * LocalDate b = LocalDateTime.of(2012, 7, 1, 12, 00);
1900 * a.isEqual(b) == false
1901 * a.isEqual(a) == true
1902 * b.isEqual(a) == false
1903 * </pre>
1904 * <p>
1905 * This method only considers the position of the two date-times on the local time-line.
1906 * It does not take into account the chronology, or calendar system.
1907 * This is different from the comparison in {@link #compareTo(ChronoLocalDateTime)},
1908 * but is the same approach as {@link ChronoLocalDateTime#timeLineOrder()}.
1909 *
1910 * @param other the other date-time to compare to, not null
1911 * @return true if this date-time is equal to the specified date-time
1912 */
1913 @Override // override for Javadoc and performance
1914 public boolean isEqual(ChronoLocalDateTime<?> other) {
1915     if (other instanceof LocalDateTime) {
1916         return compareTo((LocalDateTime) other) == 0;
1917     }
1918     return ChronoLocalDateTime.super.isEqual(other);
1919 }
1920
1921 //-----
1922 /**
1923 * Checks if this date-time is equal to another date-time.

```

```

1924 * <p>
1925 * Compares this {@code LocalDateTime} with another ensuring that the date-time is the same.
1926 * Only objects of type {@code LocalDateTime} are compared, other types return false.
1927 *
1928 * @param obj the object to check, null returns false
1929 * @return true if this is equal to the other date-time
1930 */
1931 @Override
1932 public boolean equals(Object obj) {
1933     if (this == obj) {
1934         return true;
1935     }
1936     if (obj instanceof LocalDateTime) {
1937         LocalDateTime other = (LocalDateTime) obj;
1938         return date.equals(other.date) && time.equals(other.time);
1939     }
1940     return false;
1941 }
1942
1943 /**
1944 * A hash code for this date-time.
1945 *
1946 * @return a suitable hash code
1947 */
1948 @Override
1949 public int hashCode() {
1950     return date.hashCode() ^ time.hashCode();
1951 }
1952
1953 //-----
1954 /**
1955 * Outputs this date-time as a {@code String}, such as {@code 2007-12-03T10:15:30}.
1956 * <p>
1957 * The output will be one of the following ISO-8601 formats:
1958 * <ul>
1959 * <li>{@code yyyy-MM-dd'T'HH:mm}</li>
1960 * <li>{@code yyyy-MM-dd'T'HH:mm:ss}</li>
1961 * <li>{@code yyyy-MM-dd'T'HH:mm:ss.SSS}</li>
1962 * <li>{@code yyyy-MM-dd'T'HH:mm:ss.SSSSSS}</li>
1963 * <li>{@code yyyy-MM-dd'T'HH:mm:ss.SSSSSSSS}</li>
1964 * </ul>
1965 * The format used will be the shortest that outputs the full value of
1966 * the time where the omitted parts are implied to be zero.
1967 *
1968 * @return a string representation of this date-time, not null
1969 */
1970 @Override
1971 public String toString() {
1972     return date.toString() + 'T' + time.toString();
1973 }
1974
1975 //-----
1976 /**

```

```

1977     * Writes the object using a
1978     * <a href="../../serialized-form.html#java.time.Ser">dedicated serialized form</a>.
1979     * @serialData
1980     * <pre>
1981     *   out.writeByte(5); // identifies a LocalDateTime
1982     *   // the <a href="../../serialized-form.html#java.time.LocalDate">date</a> excluding the one
1983     *       byte header
1984     *   // the <a href="../../serialized-form.html#java.time.LocalTime">time</a> excluding the one
1985     *       byte header
1986     * </pre>
1987     *
1988     * @return the instance of {@code Ser}, not null
1989     */
1990     private Object writeReplace() {
1991         return new Ser(Ser.LOCAL_DATE_TIME_TYPE, this);
1992     }
1993
1994     /**
1995     * Defend against malicious streams.
1996     *
1997     * @param s the stream to read
1998     * @throws InvalidObjectException always
1999     */
2000     private void readObject(ObjectInputStream s) throws InvalidObjectException {
2001         throw new InvalidObjectException("Deserialization via serialization delegate");
2002     }
2003
2004     void writeExternal(DataOutput out) throws IOException {
2005         date.writeExternal(out);
2006         time.writeExternal(out);
2007     }
2008
2009     static LocalDateTime readExternal(DataInput in) throws IOException {
2010         LocalDate date = LocalDate.readExternal(in);
2011         LocalTime time = LocalTime.readExternal(in);
2012         return LocalDateTime.of(date, time);
2013     }
2014 }

```

4.8 LocalTime.java

Secuencia 57: java/time/LocalTime.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *

```

```
11  *
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26 /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62 package java.time;
63
```

```
64 import static java.time.temporal.ChronoField.HOUR_OF_DAY;
65 import static java.time.temporal.ChronoField.MICRO_OF_DAY;
66 import static java.time.temporal.ChronoField.MINUTE_OF_HOUR;
67 import static java.time.temporal.ChronoField.NANO_OF_DAY;
68 import static java.time.temporal.ChronoField.NANO_OF_SECOND;
69 import static java.time.temporal.ChronoField.SECOND_OF_DAY;
70 import static java.time.temporal.ChronoField.SECOND_OF_MINUTE;
71 import static java.time.temporal.ChronoUnit.NANOS;
72
73 import java.io.DataInput;
74 import java.io.DataOutput;
75 import java.io.IOException;
76 import java.io.InvalidObjectException;
77 import java.io.ObjectInputStream;
78 import java.io.Serializable;
79 import java.time.format.DateTimeFormatter;
80 import java.time.format.DateTimeParseException;
81 import java.time.temporal.ChronoField;
82 import java.time.temporal.ChronoUnit;
83 import java.time.temporal.Temporal;
84 import java.time.temporal.TemporalAccessor;
85 import java.time.temporal.TemporalAdjuster;
86 import java.time.temporal.TemporalAmount;
87 import java.time.temporal.TemporalField;
88 import java.time.temporal.TemporalQueries;
89 import java.time.temporal.TemporalQuery;
90 import java.time.temporal.TemporalUnit;
91 import java.time.temporal.UnsupportedTemporalTypeException;
92 import java.time.temporal.ValueRange;
93 import java.util.Objects;
94
95 /**
96  * A time without a time-zone in the ISO-8601 calendar system,
97  * such as {@code 10:15:30}.
98  * <p>
99  * {@code LocalTime} is an immutable date-time object that represents a time,
100  * often viewed as hour-minute-second.
101  * Time is represented to nanosecond precision.
102  * For example, the value "13:45.30.123456789" can be stored in a {@code LocalTime}.
103  * <p>
104  * This class does not store or represent a date or time-zone.
105  * Instead, it is a description of the local time as seen on a wall clock.
106  * It cannot represent an instant on the time-line without additional information
107  * such as an offset or time-zone.
108  * <p>
109  * The ISO-8601 calendar system is the modern civil calendar system used today
110  * in most of the world. This API assumes that all calendar systems use the same
111  * representation, this class, for time-of-day.
112  *
113  * <p>
114  * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
115  * class; use of identity-sensitive operations (including reference equality
116  * ({@code ==}), identity hash code, or synchronization) on instances of
```



```
117 * {@code LocalTime} may have unpredictable results and should be avoided.
118 * The {@code equals} method should be used for comparisons.
119 *
120 * @implSpec
121 * This class is immutable and thread-safe.
122 *
123 * @since 1.8
124 */
125 public final class LocalTime
126     implements Temporal, TemporalAdjuster, Comparable<LocalTime>, Serializable {
127
128     /**
129      * The minimum supported {@code LocalTime}, '00:00'.
130      * This is the time of midnight at the start of the day.
131      */
132     public static final LocalTime MIN;
133
134     /**
135      * The maximum supported {@code LocalTime}, '23:59:59.999999999'.
136      * This is the time just before midnight at the end of the day.
137      */
138     public static final LocalTime MAX;
139
140     /**
141      * The time of midnight at the start of the day, '00:00'.
142      */
143     public static final LocalTime MIDNIGHT;
144
145     /**
146      * The time of noon in the middle of the day, '12:00'.
147      */
148     public static final LocalTime NOON;
149
150     /**
151      * Constants for the local time of each hour.
152      */
153     private static final LocalTime[] HOURS = new LocalTime[24];
154     static {
155         for (int i = 0; i < HOURS.length; i++) {
156             HOURS[i] = new LocalTime(i, 0, 0, 0);
157         }
158         MIDNIGHT = HOURS[0];
159         NOON = HOURS[12];
160         MIN = HOURS[0];
161         MAX = new LocalTime(23, 59, 59, 999_999_999);
162     }
163
164     /**
165      * Hours per day.
166      */
167     static final int HOURS_PER_DAY = 24;
168
169     /**
170      * Minutes per hour.
171      */
172     static final int MINUTES_PER_HOUR = 60;
173
174     /**
175      * Minutes per day.
```

```
170     */
171     static final int MINUTES_PER_DAY = MINUTES_PER_HOUR * HOURS_PER_DAY;
172     /**
173      * Seconds per minute.
174      */
175     static final int SECONDS_PER_MINUTE = 60;
176     /**
177      * Seconds per hour.
178      */
179     static final int SECONDS_PER_HOUR = SECONDS_PER_MINUTE * MINUTES_PER_HOUR;
180     /**
181      * Seconds per day.
182      */
183     static final int SECONDS_PER_DAY = SECONDS_PER_HOUR * HOURS_PER_DAY;
184     /**
185      * Milliseconds per day.
186      */
187     static final long MILLIS_PER_DAY = SECONDS_PER_DAY * 1000L;
188     /**
189      * Microseconds per day.
190      */
191     static final long MICROS_PER_DAY = SECONDS_PER_DAY * 1000_000L;
192     /**
193      * Nanos per second.
194      */
195     static final long NANOS_PER_SECOND = 1000_000_000L;
196     /**
197      * Nanos per minute.
198      */
199     static final long NANOS_PER_MINUTE = NANOS_PER_SECOND * SECONDS_PER_MINUTE;
200     /**
201      * Nanos per hour.
202      */
203     static final long NANOS_PER_HOUR = NANOS_PER_MINUTE * MINUTES_PER_HOUR;
204     /**
205      * Nanos per day.
206      */
207     static final long NANOS_PER_DAY = NANOS_PER_HOUR * HOURS_PER_DAY;
208
209     /**
210      * Serialization version.
211      */
212     private static final long serialVersionUID = 6414437269572265201L;
213
214     /**
215      * The hour.
216      */
217     private final byte hour;
218     /**
219      * The minute.
220      */
221     private final byte minute;
222     /**
```

```

223     * The second.
224     */
225     private final byte second;
226     /**
227     * The nanosecond.
228     */
229     private final int nano;
230
231     //-----
232     /**
233     * Obtains the current time from the system clock in the default time-zone.
234     * <p>
235     * This will query the {@link Clock#systemDefaultZone() system clock} in the default
236     * time-zone to obtain the current time.
237     * <p>
238     * Using this method will prevent the ability to use an alternate clock for testing
239     * because the clock is hard-coded.
240     *
241     * @return the current time using the system clock and default time-zone, not null
242     */
243     public static LocalTime now() {
244         return now(Clock.systemDefaultZone());
245     }
246
247     /**
248     * Obtains the current time from the system clock in the specified time-zone.
249     * <p>
250     * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current time.
251     * Specifying the time-zone avoids dependence on the default time-zone.
252     * <p>
253     * Using this method will prevent the ability to use an alternate clock for testing
254     * because the clock is hard-coded.
255     *
256     * @param zone the zone ID to use, not null
257     * @return the current time using the system clock, not null
258     */
259     public static LocalTime now(ZoneId zone) {
260         return now(Clock.system(zone));
261     }
262
263     /**
264     * Obtains the current time from the specified clock.
265     * <p>
266     * This will query the specified clock to obtain the current time.
267     * Using this method allows the use of an alternate clock for testing.
268     * The alternate clock may be introduced using {@link Clock dependency injection}.
269     *
270     * @param clock the clock to use, not null
271     * @return the current time, not null
272     */
273     public static LocalTime now(Clock clock) {
274         Objects.requireNonNull(clock, "clock");
275         // inline OffsetTime factory to avoid creating object and InstantProvider checks

```

```

276     final Instant now = clock.instant(); // called once
277     ZoneOffset offset = clock.getZone().getRules().getOffset(now);
278     long localSecond = now.getEpochSecond() + offset.getTotalSeconds(); // overflow caught
        later
279     int secsOfDay = (int) Math.floorMod(localSecond, SECONDS_PER_DAY);
280     return ofNanoOfDay(secsOfDay * NANOS_PER_SECOND + now.getNano());
281 }
282
283 //-----
284 /**
285  * Obtains an instance of {@code LocalTime} from an hour and minute.
286  * <p>
287  * This returns a {@code LocalTime} with the specified hour and minute.
288  * The second and nanosecond fields will be set to zero.
289  *
290  * @param hour the hour-of-day to represent, from 0 to 23
291  * @param minute the minute-of-hour to represent, from 0 to 59
292  * @return the local time, not null
293  * @throws DateTimeException if the value of any field is out of range
294  */
295 public static LocalTime of(int hour, int minute) {
296     HOUR_OF_DAY.checkValidValue(hour);
297     if (minute == 0) {
298         return HOURS[hour]; // for performance
299     }
300     MINUTE_OF_HOUR.checkValidValue(minute);
301     return new LocalTime(hour, minute, 0, 0);
302 }
303
304 /**
305  * Obtains an instance of {@code LocalTime} from an hour, minute and second.
306  * <p>
307  * This returns a {@code LocalTime} with the specified hour, minute and second.
308  * The nanosecond field will be set to zero.
309  *
310  * @param hour the hour-of-day to represent, from 0 to 23
311  * @param minute the minute-of-hour to represent, from 0 to 59
312  * @param second the second-of-minute to represent, from 0 to 59
313  * @return the local time, not null
314  * @throws DateTimeException if the value of any field is out of range
315  */
316 public static LocalTime of(int hour, int minute, int second) {
317     HOUR_OF_DAY.checkValidValue(hour);
318     if ((minute | second) == 0) {
319         return HOURS[hour]; // for performance
320     }
321     MINUTE_OF_HOUR.checkValidValue(minute);
322     SECOND_OF_MINUTE.checkValidValue(second);
323     return new LocalTime(hour, minute, second, 0);
324 }
325
326 /**
327  * Obtains an instance of {@code LocalTime} from an hour, minute, second and nanosecond.

```

```

328 * <p>
329 * This returns a {@code LocalTime} with the specified hour, minute, second and nanosecond.
330 *
331 * @param hour the hour-of-day to represent, from 0 to 23
332 * @param minute the minute-of-hour to represent, from 0 to 59
333 * @param second the second-of-minute to represent, from 0 to 59
334 * @param nanoOfSecond the nano-of-second to represent, from 0 to 999,999,999
335 * @return the local time, not null
336 * @throws DateTimeException if the value of any field is out of range
337 */
338 public static LocalTime of(int hour, int minute, int second, int nanoOfSecond) {
339     HOUR_OF_DAY.checkValidValue(hour);
340     MINUTE_OF_HOUR.checkValidValue(minute);
341     SECOND_OF_MINUTE.checkValidValue(second);
342     NANO_OF_SECOND.checkValidValue(nanoOfSecond);
343     return create(hour, minute, second, nanoOfSecond);
344 }
345
346 //-----
347 /**
348 * Obtains an instance of {@code LocalTime} from a second-of-day value.
349 * <p>
350 * This returns a {@code LocalTime} with the specified second-of-day.
351 * The nanosecond field will be set to zero.
352 *
353 * @param secondOfDay the second-of-day, from {@code 0} to {@code 24 * 60 * 60 - 1}
354 * @return the local time, not null
355 * @throws DateTimeException if the second-of-day value is invalid
356 */
357 public static LocalTime ofSecondOfDay(long secondOfDay) {
358     SECOND_OF_DAY.checkValidValue(secondOfDay);
359     int hours = (int) (secondOfDay / SECONDS_PER_HOUR);
360     secondOfDay -= hours * SECONDS_PER_HOUR;
361     int minutes = (int) (secondOfDay / SECONDS_PER_MINUTE);
362     secondOfDay -= minutes * SECONDS_PER_MINUTE;
363     return create(hours, minutes, (int) secondOfDay, 0);
364 }
365
366 /**
367 * Obtains an instance of {@code LocalTime} from a nanos-of-day value.
368 * <p>
369 * This returns a {@code LocalTime} with the specified nanosecond-of-day.
370 *
371 * @param nanoOfDay the nano of day, from {@code 0} to {@code 24 * 60 * 60 * 1,000,000,000 - 1}
372 * @return the local time, not null
373 * @throws DateTimeException if the nanos of day value is invalid
374 */
375 public static LocalTime ofNanoOfDay(long nanoOfDay) {
376     NANO_OF_DAY.checkValidValue(nanoOfDay);
377     int hours = (int) (nanoOfDay / NANOS_PER_HOUR);
378     nanoOfDay -= hours * NANOS_PER_HOUR;
379     int minutes = (int) (nanoOfDay / NANOS_PER_MINUTE);

```

```

380     nanoOfDay -= minutes * NANOS_PER_MINUTE;
381     int seconds = (int) (nanoOfDay / NANOS_PER_SECOND);
382     nanoOfDay -= seconds * NANOS_PER_SECOND;
383     return create(hours, minutes, seconds, (int) nanoOfDay);
384 }
385
386 //-----
387 /**
388  * Obtains an instance of {@code LocalTime} from a temporal object.
389  * <p>
390  * This obtains a local time based on the specified temporal.
391  * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
392  * which this factory converts to an instance of {@code LocalTime}.
393  * <p>
394  * The conversion uses the {@link TemporalQueries#localTime()} query, which relies
395  * on extracting the {@link ChronoField#NANO_OF_DAY NANO_OF_DAY} field.
396  * <p>
397  * This method matches the signature of the functional interface {@link TemporalQuery}
398  * allowing it to be used as a query via method reference, {@code LocalTime::from}.
399  *
400  * @param temporal the temporal object to convert, not null
401  * @return the local time, not null
402  * @throws DateTimeException if unable to convert to a {@code LocalTime}
403  */
404 public static LocalTime from(TemporalAccessor temporal) {
405     Objects.requireNonNull(temporal, "temporal");
406     LocalTime time = temporal.query(TemporalQueries.localTime());
407     if (time == null) {
408         throw new DateTimeException("Unable to obtain LocalTime from TemporalAccessor: " +
409             temporal + " of type " + temporal.getClass().getName());
410     }
411     return time;
412 }
413
414 //-----
415 /**
416  * Obtains an instance of {@code LocalTime} from a text string such as {@code 10:15}.
417  * <p>
418  * The string must represent a valid time and is parsed using
419  * {@link java.time.format.DateTimeFormatter#ISO_LOCAL_TIME}.
420  *
421  * @param text the text to parse such as "10:15:30", not null
422  * @return the parsed local time, not null
423  * @throws DateTimeParseException if the text cannot be parsed
424  */
425 public static LocalTime parse(CharSequence text) {
426     return parse(text, DateTimeFormatter.ISO_LOCAL_TIME);
427 }
428
429 /**
430  * Obtains an instance of {@code LocalTime} from a text string using a specific formatter.
431  * <p>
432  * The text is parsed using the formatter, returning a time.

```

```

433     *
434     * @param text the text to parse, not null
435     * @param formatter the formatter to use, not null
436     * @return the parsed local time, not null
437     * @throws DateTimeParseException if the text cannot be parsed
438     */
439     public static LocalTime parse(CharSequence text, DateTimeFormatter formatter) {
440         Objects.requireNonNull(formatter, "formatter");
441         return formatter.parse(text, LocalTime::from);
442     }
443
444     //-----
445     /**
446     * Creates a local time from the hour, minute, second and nanosecond fields.
447     * <p>
448     * This factory may return a cached value, but applications must not rely on this.
449     *
450     * @param hour the hour-of-day to represent, validated from 0 to 23
451     * @param minute the minute-of-hour to represent, validated from 0 to 59
452     * @param second the second-of-minute to represent, validated from 0 to 59
453     * @param nanoOfSecond the nano-of-second to represent, validated from 0 to 999,999,999
454     * @return the local time, not null
455     */
456     private static LocalTime create(int hour, int minute, int second, int nanoOfSecond) {
457         if ((minute | second | nanoOfSecond) == 0) {
458             return HOURS[hour];
459         }
460         return new LocalTime(hour, minute, second, nanoOfSecond);
461     }
462
463     /**
464     * Constructor, previously validated.
465     *
466     * @param hour the hour-of-day to represent, validated from 0 to 23
467     * @param minute the minute-of-hour to represent, validated from 0 to 59
468     * @param second the second-of-minute to represent, validated from 0 to 59
469     * @param nanoOfSecond the nano-of-second to represent, validated from 0 to 999,999,999
470     */
471     private LocalTime(int hour, int minute, int second, int nanoOfSecond) {
472         this.hour = (byte) hour;
473         this.minute = (byte) minute;
474         this.second = (byte) second;
475         this.nano = nanoOfSecond;
476     }
477
478     //-----
479     /**
480     * Checks if the specified field is supported.
481     * <p>
482     * This checks if this time can be queried for the specified field.
483     * If false, then calling the {@link #range(TemporalField) range},
484     * {@link #get(TemporalField) get} and {@link #with(TemporalField, long)}
485     * methods will throw an exception.

```

```

486 * <p>
487 * If the field is a {@link ChronoField} then the query is implemented here.
488 * The supported fields are:
489 * <ul>
490 * <li>{@code NANO_OF_SECOND}
491 * <li>{@code NANO_OF_DAY}
492 * <li>{@code MICRO_OF_SECOND}
493 * <li>{@code MICRO_OF_DAY}
494 * <li>{@code MILLI_OF_SECOND}
495 * <li>{@code MILLI_OF_DAY}
496 * <li>{@code SECOND_OF_MINUTE}
497 * <li>{@code SECOND_OF_DAY}
498 * <li>{@code MINUTE_OF_HOUR}
499 * <li>{@code MINUTE_OF_DAY}
500 * <li>{@code HOUR_OF_AMPM}
501 * <li>{@code CLOCK_HOUR_OF_AMPM}
502 * <li>{@code HOUR_OF_DAY}
503 * <li>{@code CLOCK_HOUR_OF_DAY}
504 * <li>{@code AMPM_OF_DAY}
505 * </ul>
506 * All other {@code ChronoField} instances will return false.
507 * <p>
508 * If the field is not a {@code ChronoField}, then the result of this method
509 * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
510 * passing {@code this} as the argument.
511 * Whether the field is supported is determined by the field.
512 *
513 * @param field the field to check, null returns false
514 * @return true if the field is supported on this time, false if not
515 */
516 @Override
517 public boolean isSupported(TemporalField field) {
518     if (field instanceof ChronoField) {
519         return field.isTimeBased();
520     }
521     return field != null && field.isSupportedBy(this);
522 }
523
524 /**
525 * Checks if the specified unit is supported.
526 * <p>
527 * This checks if the specified unit can be added to, or subtracted from, this time.
528 * If false, then calling the {@link #plus(long, TemporalUnit)} and
529 * {@link #minus(long, TemporalUnit) minus} methods will throw an exception.
530 * <p>
531 * If the unit is a {@link ChronoUnit} then the query is implemented here.
532 * The supported units are:
533 * <ul>
534 * <li>{@code NANOS}
535 * <li>{@code MICROS}
536 * <li>{@code MILLIS}
537 * <li>{@code SECONDS}
538 * <li>{@code MINUTES}

```



```

539     * <li>{@code HOURS}
540     * <li>{@code HALF_DAYS}
541     * </ul>
542     * All other {@code ChronoUnit} instances will return false.
543     * <p>
544     * If the unit is not a {@code ChronoUnit}, then the result of this method
545     * is obtained by invoking {@code TemporalUnit.isSupportedBy(Temporal)}
546     * passing {@code this} as the argument.
547     * Whether the unit is supported is determined by the unit.
548     *
549     * @param unit the unit to check, null returns false
550     * @return true if the unit can be added/subtracted, false if not
551     */
552     @Override // override for Javadoc
553     public boolean isSupported(TemporalUnit unit) {
554         if (unit instanceof ChronoUnit) {
555             return unit.isTimeBased();
556         }
557         return unit != null && unit.isSupportedBy(this);
558     }
559
560     //-----
561     /**
562     * Gets the range of valid values for the specified field.
563     * <p>
564     * The range object expresses the minimum and maximum valid values for a field.
565     * This time is used to enhance the accuracy of the returned range.
566     * If it is not possible to return the range, because the field is not supported
567     * or for some other reason, an exception is thrown.
568     * <p>
569     * If the field is a {@link ChronoField} then the query is implemented here.
570     * The {@link #isSupported(TemporalField) supported fields} will return
571     * appropriate range instances.
572     * All other {@code ChronoField} instances will throw an {@code
573         UnsupportedOperationException}.
574     * <p>
575     * If the field is not a {@code ChronoField}, then the result of this method
576     * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
577     * passing {@code this} as the argument.
578     * Whether the range can be obtained is determined by the field.
579     *
580     * @param field the field to query the range for, not null
581     * @return the range of valid values for the field, not null
582     * @throws DateTimeException if the range for the field cannot be obtained
583     * @throws UnsupportedOperationException if the field is not supported
584     */
585     @Override // override for Javadoc
586     public ValueRange range(TemporalField field) {
587         return Temporal.super.range(field);
588     }
589
590     /**
591     * Gets the value of the specified field from this time as an {@code int}.

```

```

591 * <p>
592 * This queries this time for the value of the specified field.
593 * The returned value will always be within the valid range of values for the field.
594 * If it is not possible to return the value, because the field is not supported
595 * or for some other reason, an exception is thrown.
596 * <p>
597 * If the field is a {@link ChronoField} then the query is implemented here.
598 * The {@link #isSupported(TemporalField) supported fields} will return valid
599 * values based on this time, except {@code NANO_OF_DAY} and {@code MICRO_OF_DAY}
600 * which are too large to fit in an {@code int} and throw a {@code DateTimeException}.
601 * All other {@code ChronoField} instances will throw an {@code
    UnsupportedTemporalTypeException}.
602 * <p>
603 * If the field is not a {@code ChronoField}, then the result of this method
604 * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
605 * passing {@code this} as the argument. Whether the value can be obtained,
606 * and what the value represents, is determined by the field.
607 *
608 * @param field the field to get, not null
609 * @return the value for the field
610 * @throws DateTimeException if a value for the field cannot be obtained or
611 *         the value is outside the range of valid values for the field
612 * @throws UnsupportedTemporalTypeException if the field is not supported or
613 *         the range of values exceeds an {@code int}
614 * @throws ArithmeticException if numeric overflow occurs
615 */
616 @Override // override for Javadoc and performance
617 public int get(TemporalField field) {
618     if (field instanceof ChronoField) {
619         return get0(field);
620     }
621     return Temporal.super.get(field);
622 }
623
624 /**
625 * Gets the value of the specified field from this time as a {@code long}.
626 * <p>
627 * This queries this time for the value of the specified field.
628 * If it is not possible to return the value, because the field is not supported
629 * or for some other reason, an exception is thrown.
630 * <p>
631 * If the field is a {@link ChronoField} then the query is implemented here.
632 * The {@link #isSupported(TemporalField) supported fields} will return valid
633 * values based on this time.
634 * All other {@code ChronoField} instances will throw an {@code
    UnsupportedTemporalTypeException}.
635 * <p>
636 * If the field is not a {@code ChronoField}, then the result of this method
637 * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
638 * passing {@code this} as the argument. Whether the value can be obtained,
639 * and what the value represents, is determined by the field.
640 *
641 * @param field the field to get, not null

```

```

642     * @return the value for the field
643     * @throws DateTimeException if a value for the field cannot be obtained
644     * @throws UnsupportedTemporalTypeException if the field is not supported
645     * @throws ArithmeticException if numeric overflow occurs
646     */
647     @Override
648     public long getLong(TemporalField field) {
649         if (field instanceof ChronoField) {
650             if (field == NANO_OF_DAY) {
651                 return toNanoOfDay();
652             }
653             if (field == MICRO_OF_DAY) {
654                 return toNanoOfDay() / 1000;
655             }
656             return get0(field);
657         }
658         return field.getFrom(this);
659     }
660
661     private int get0(TemporalField field) {
662         switch ((ChronoField) field) {
663             case NANO_OF_SECOND: return nano;
664             case NANO_OF_DAY: throw new UnsupportedTemporalTypeException("Invalid field 'NanoOfDay'
665                                     for get() method, use getLong() instead");
666             case MICRO_OF_SECOND: return nano / 1000;
667             case MICRO_OF_DAY: throw new UnsupportedTemporalTypeException("Invalid field '
668                                     MicroOfDay' for get() method, use getLong() instead");
669             case MILLI_OF_SECOND: return nano / 1000_000;
670             case MILLI_OF_DAY: return (int) (toNanoOfDay() / 1000_000);
671             case SECOND_OF_MINUTE: return second;
672             case SECOND_OF_DAY: return toSecondOfDay();
673             case MINUTE_OF_HOUR: return minute;
674             case MINUTE_OF_DAY: return hour * 60 + minute;
675             case HOUR_OF_AMPM: return hour % 12;
676             case CLOCK_HOUR_OF_AMPM: int ham = hour % 12; return (ham % 12 == 0 ? 12 : ham);
677             case HOUR_OF_DAY: return hour;
678             case CLOCK_HOUR_OF_DAY: return (hour == 0 ? 24 : hour);
679             case AMPM_OF_DAY: return hour / 12;
680         }
681         throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
682     }
683
684     //-----
685     /**
686     * Gets the hour-of-day field.
687     *
688     * @return the hour-of-day, from 0 to 23
689     */
690     public int getHour() {
691         return hour;
692     }
693
694     /**

```

```

693     * Gets the minute-of-hour field.
694     *
695     * @return the minute-of-hour, from 0 to 59
696     */
697     public int getMinute() {
698         return minute;
699     }
700
701     /**
702     * Gets the second-of-minute field.
703     *
704     * @return the second-of-minute, from 0 to 59
705     */
706     public int getSecond() {
707         return second;
708     }
709
710     /**
711     * Gets the nano-of-second field.
712     *
713     * @return the nano-of-second, from 0 to 999,999,999
714     */
715     public int getNano() {
716         return nano;
717     }
718
719     //-----
720     /**
721     * Returns an adjusted copy of this time.
722     * <p>
723     * This returns a {@code LocalTime}, based on this one, with the time adjusted.
724     * The adjustment takes place using the specified adjuster strategy object.
725     * Read the documentation of the adjuster to understand what adjustment will be made.
726     * <p>
727     * A simple adjuster might simply set the one of the fields, such as the hour field.
728     * A more complex adjuster might set the time to the last hour of the day.
729     * <p>
730     * The result of this method is obtained by invoking the
731     * {@link TemporalAdjuster#adjustInto(Temporal)} method on the
732     * specified adjuster passing {@code this} as the argument.
733     * <p>
734     * This instance is immutable and unaffected by this method call.
735     *
736     * @param adjuster the adjuster to use, not null
737     * @return a {@code LocalTime} based on {@code this} with the adjustment made, not null
738     * @throws DateTimeException if the adjustment cannot be made
739     * @throws ArithmeticException if numeric overflow occurs
740     */
741     @Override
742     public LocalTime with(TemporalAdjuster adjuster) {
743         // optimizations
744         if (adjuster instanceof LocalTime) {
745             return (LocalTime) adjuster;

```

```

746     }
747     return (LocalTime) adjuster.adjustInto(this);
748 }
749
750 /**
751  * Returns a copy of this time with the specified field set to a new value.
752  * <p>
753  * This returns a {@code LocalTime}, based on this one, with the value
754  * for the specified field changed.
755  * This can be used to change any supported field, such as the hour, minute or second.
756  * If it is not possible to set the value, because the field is not supported or for
757  * some other reason, an exception is thrown.
758  * <p>
759  * If the field is a {@link ChronoField} then the adjustment is implemented here.
760  * The supported fields behave as follows:
761  * <ul>
762  * <li>{@code NANO_OF_SECOND} -
763  * Returns a {@code LocalTime} with the specified nano-of-second.
764  * The hour, minute and second will be unchanged.
765  * <li>{@code NANO_OF_DAY} -
766  * Returns a {@code LocalTime} with the specified nano-of-day.
767  * This completely replaces the time and is equivalent to {@link #ofNanoOfDay(long)}.
768  * <li>{@code MICRO_OF_SECOND} -
769  * Returns a {@code LocalTime} with the nano-of-second replaced by the specified
770  * micro-of-second multiplied by 1,000.
771  * The hour, minute and second will be unchanged.
772  * <li>{@code MICRO_OF_DAY} -
773  * Returns a {@code LocalTime} with the specified micro-of-day.
774  * This completely replaces the time and is equivalent to using {@link #ofNanoOfDay(long)}
775  * with the micro-of-day multiplied by 1,000.
776  * <li>{@code MILLI_OF_SECOND} -
777  * Returns a {@code LocalTime} with the nano-of-second replaced by the specified
778  * milli-of-second multiplied by 1,000,000.
779  * The hour, minute and second will be unchanged.
780  * <li>{@code MILLI_OF_DAY} -
781  * Returns a {@code LocalTime} with the specified milli-of-day.
782  * This completely replaces the time and is equivalent to using {@link #ofNanoOfDay(long)}
783  * with the milli-of-day multiplied by 1,000,000.
784  * <li>{@code SECOND_OF_MINUTE} -
785  * Returns a {@code LocalTime} with the specified second-of-minute.
786  * The hour, minute and nano-of-second will be unchanged.
787  * <li>{@code SECOND_OF_DAY} -
788  * Returns a {@code LocalTime} with the specified second-of-day.
789  * The nano-of-second will be unchanged.
790  * <li>{@code MINUTE_OF_HOUR} -
791  * Returns a {@code LocalTime} with the specified minute-of-hour.
792  * The hour, second-of-minute and nano-of-second will be unchanged.
793  * <li>{@code MINUTE_OF_DAY} -
794  * Returns a {@code LocalTime} with the specified minute-of-day.
795  * The second-of-minute and nano-of-second will be unchanged.
796  * <li>{@code HOUR_OF_AMPM} -
797  * Returns a {@code LocalTime} with the specified hour-of-am-pm.
798  * The AM/PM, minute-of-hour, second-of-minute and nano-of-second will be unchanged.

```

```

799 * <li>{@code CLOCK_HOUR_OF_AMPM} -
800 * Returns a {@code LocalTime} with the specified clock-hour-of-am-pm.
801 * The AM/PM, minute-of-hour, second-of-minute and nano-of-second will be unchanged.
802 * <li>{@code HOUR_OF_DAY} -
803 * Returns a {@code LocalTime} with the specified hour-of-day.
804 * The minute-of-hour, second-of-minute and nano-of-second will be unchanged.
805 * <li>{@code CLOCK_HOUR_OF_DAY} -
806 * Returns a {@code LocalTime} with the specified clock-hour-of-day.
807 * The minute-of-hour, second-of-minute and nano-of-second will be unchanged.
808 * <li>{@code AMPM_OF_DAY} -
809 * Returns a {@code LocalTime} with the specified AM/PM.
810 * The hour-of-am-pm, minute-of-hour, second-of-minute and nano-of-second will be unchanged.
811 * </ul>
812 * <p>
813 * In all cases, if the new value is outside the valid range of values for the field
814 * then a {@code DateTimeException} will be thrown.
815 * <p>
816 * All other {@code ChronoField} instances will throw an {@code
      UnsupportedTemporalTypeException}.
817 * <p>
818 * If the field is not a {@code ChronoField}, then the result of this method
819 * is obtained by invoking {@code TemporalField.adjustInto(Temporal, long)}
820 * passing {@code this} as the argument. In this case, the field determines
821 * whether and how to adjust the instant.
822 * <p>
823 * This instance is immutable and unaffected by this method call.
824 *
825 * @param field the field to set in the result, not null
826 * @param newValue the new value of the field in the result
827 * @return a {@code LocalTime} based on {@code this} with the specified field set, not null
828 * @throws DateTimeException if the field cannot be set
829 * @throws UnsupportedTemporalTypeException if the field is not supported
830 * @throws ArithmeticException if numeric overflow occurs
831 */
832 @Override
833 public LocalTime with(TemporalField field, long newValue) {
834     if (field instanceof ChronoField) {
835         ChronoField f = (ChronoField) field;
836         f.checkValidValue(newValue);
837         switch (f) {
838             case NANO_OF_SECOND: return withNano((int) newValue);
839             case NANO_OF_DAY: return LocalTime.ofNanoOfDay(newValue);
840             case MICRO_OF_SECOND: return withNano((int) newValue * 1000);
841             case MICRO_OF_DAY: return LocalTime.ofNanoOfDay(newValue * 1000);
842             case MILLI_OF_SECOND: return withNano((int) newValue * 1000_000);
843             case MILLI_OF_DAY: return LocalTime.ofNanoOfDay(newValue * 1000_000);
844             case SECOND_OF_MINUTE: return withSecond((int) newValue);
845             case SECOND_OF_DAY: return plusSeconds(newValue - toSecondOfDay());
846             case MINUTE_OF_HOUR: return withMinute((int) newValue);
847             case MINUTE_OF_DAY: return plusMinutes(newValue - (hour * 60 + minute));
848             case HOUR_OF_AMPM: return plusHours(newValue - (hour % 12));
849             case CLOCK_HOUR_OF_AMPM: return plusHours((newValue == 12 ? 0 : newValue) - (hour %
      12));

```

```

850         case HOUR_OF_DAY: return withHour((int) newValue);
851         case CLOCK_HOUR_OF_DAY: return withHour((int) (newValue == 24 ? 0 : newValue));
852         case AMPM_OF_DAY: return plusHours((newValue - (hour / 12)) * 12);
853     }
854     throw new UnsupportedOperationException("Unsupported field: " + field);
855 }
856 return field.adjustInto(this, newValue);
857 }
858
859 //-----
860 /**
861  * Returns a copy of this {@code LocalTime} with the hour-of-day altered.
862  * <p>
863  * This instance is immutable and unaffected by this method call.
864  *
865  * @param hour the hour-of-day to set in the result, from 0 to 23
866  * @return a {@code LocalTime} based on this time with the requested hour, not null
867  * @throws DateTimeException if the hour value is invalid
868  */
869 public LocalTime withHour(int hour) {
870     if (this.hour == hour) {
871         return this;
872     }
873     HOUR_OF_DAY.checkValidValue(hour);
874     return create(hour, minute, second, nano);
875 }
876
877 /**
878  * Returns a copy of this {@code LocalTime} with the minute-of-hour altered.
879  * <p>
880  * This instance is immutable and unaffected by this method call.
881  *
882  * @param minute the minute-of-hour to set in the result, from 0 to 59
883  * @return a {@code LocalTime} based on this time with the requested minute, not null
884  * @throws DateTimeException if the minute value is invalid
885  */
886 public LocalTime withMinute(int minute) {
887     if (this.minute == minute) {
888         return this;
889     }
890     MINUTE_OF_HOUR.checkValidValue(minute);
891     return create(hour, minute, second, nano);
892 }
893
894 /**
895  * Returns a copy of this {@code LocalTime} with the second-of-minute altered.
896  * <p>
897  * This instance is immutable and unaffected by this method call.
898  *
899  * @param second the second-of-minute to set in the result, from 0 to 59
900  * @return a {@code LocalTime} based on this time with the requested second, not null
901  * @throws DateTimeException if the second value is invalid
902  */

```



```

903 public LocalTime withSecond(int second) {
904     if (this.second == second) {
905         return this;
906     }
907     SECOND_OF_MINUTE.checkValidValue(second);
908     return create(hour, minute, second, nano);
909 }
910
911 /**
912  * Returns a copy of this {@code LocalTime} with the nano-of-second altered.
913  * <p>
914  * This instance is immutable and unaffected by this method call.
915  *
916  * @param nanoOfSecond the nano-of-second to set in the result, from 0 to 999,999,999
917  * @return a {@code LocalTime} based on this time with the requested nanosecond, not null
918  * @throws DateTimeException if the nanos value is invalid
919  */
920 public LocalTime withNano(int nanoOfSecond) {
921     if (this.nano == nanoOfSecond) {
922         return this;
923     }
924     NANO_OF_SECOND.checkValidValue(nanoOfSecond);
925     return create(hour, minute, second, nanoOfSecond);
926 }
927
928 //-----
929 /**
930  * Returns a copy of this {@code LocalTime} with the time truncated.
931  * <p>
932  * Truncation returns a copy of the original time with fields
933  * smaller than the specified unit set to zero.
934  * For example, truncating with the {@link ChronoUnit#MINUTES minutes} unit
935  * will set the second-of-minute and nano-of-second field to zero.
936  * <p>
937  * The unit must have a {@linkplain TemporalUnit#getDuration() duration}
938  * that divides into the length of a standard day without remainder.
939  * This includes all supplied time units on {@link ChronoUnit} and
940  * {@link ChronoUnit#DAYS DAYS}. Other units throw an exception.
941  * <p>
942  * This instance is immutable and unaffected by this method call.
943  *
944  * @param unit the unit to truncate to, not null
945  * @return a {@code LocalTime} based on this time with the time truncated, not null
946  * @throws DateTimeException if unable to truncate
947  * @throws UnsupportedTemporalTypeException if the unit is not supported
948  */
949 public LocalTime truncatedTo(TemporalUnit unit) {
950     if (unit == ChronoUnit.NANOS) {
951         return this;
952     }
953     Duration unitDur = unit.getDuration();
954     if (unitDur.getSeconds() > SECONDS_PER_DAY) {
955         throw new UnsupportedTemporalTypeException("Unit is too large to be used for truncation

```



```

        ");
956     }
957     long dur = unitDur.toNanos();
958     if ((NANOS_PER_DAY % dur) != 0) {
959         throw new UnsupportedOperationException("Unit must divide into a standard day
            without remainder");
960     }
961     long nod = toNanoOfDay();
962     return ofNanoOfDay((nod / dur) * dur);
963 }
964
965 //-----
966 /**
967  * Returns a copy of this time with the specified amount added.
968  * <p>
969  * This returns a {@code LocalTime}, based on this one, with the specified amount added.
970  * The amount is typically {@link Duration} but may be any other type implementing
971  * the {@link TemporalAmount} interface.
972  * <p>
973  * The calculation is delegated to the amount object by calling
974  * {@link TemporalAmount#addTo(Temporal)}. The amount implementation is free
975  * to implement the addition in any way it wishes, however it typically
976  * calls back to {@link #plus(long, TemporalUnit)}. Consult the documentation
977  * of the amount implementation to determine if it can be successfully added.
978  * <p>
979  * This instance is immutable and unaffected by this method call.
980  *
981  * @param amountToAdd the amount to add, not null
982  * @return a {@code LocalTime} based on this time with the addition made, not null
983  * @throws DateTimeException if the addition cannot be made
984  * @throws ArithmeticException if numeric overflow occurs
985  */
986 @Override
987 public LocalTime plus(TemporalAmount amountToAdd) {
988     return (LocalTime) amountToAdd.addTo(this);
989 }
990
991 /**
992  * Returns a copy of this time with the specified amount added.
993  * <p>
994  * This returns a {@code LocalTime}, based on this one, with the amount
995  * in terms of the unit added. If it is not possible to add the amount, because the
996  * unit is not supported or for some other reason, an exception is thrown.
997  * <p>
998  * If the field is a {@link ChronoUnit} then the addition is implemented here.
999  * The supported fields behave as follows:
1000  * <ul>
1001  * <li>{@code NANOS} -
1002  * Returns a {@code LocalTime} with the specified number of nanoseconds added.
1003  * This is equivalent to {@link #plusNanos(long)}.
1004  * <li>{@code MICROS} -
1005  * Returns a {@code LocalTime} with the specified number of microseconds added.
1006  * This is equivalent to {@link #plusNanos(long)} with the amount

```

```

1007 * multiplied by 1,000.
1008 * <li>{@code MILLIS} -
1009 * Returns a {@code LocalTime} with the specified number of milliseconds added.
1010 * This is equivalent to {@link #plusNanos(long)} with the amount
1011 * multiplied by 1,000,000.
1012 * <li>{@code SECONDS} -
1013 * Returns a {@code LocalTime} with the specified number of seconds added.
1014 * This is equivalent to {@link #plusSeconds(long)}.
1015 * <li>{@code MINUTES} -
1016 * Returns a {@code LocalTime} with the specified number of minutes added.
1017 * This is equivalent to {@link #plusMinutes(long)}.
1018 * <li>{@code HOURS} -
1019 * Returns a {@code LocalTime} with the specified number of hours added.
1020 * This is equivalent to {@link #plusHours(long)}.
1021 * <li>{@code HALF_DAYS} -
1022 * Returns a {@code LocalTime} with the specified number of half-days added.
1023 * This is equivalent to {@link #plusHours(long)} with the amount
1024 * multiplied by 12.
1025 * </ul>
1026 * <p>
1027 * All other {@code ChronoUnit} instances will throw an {@code UnsupportedOperationException}
1028 *   }.
1029 * <p>
1030 * If the field is not a {@code ChronoUnit}, then the result of this method
1031 * is obtained by invoking {@code TemporalUnit.addTo(Temporal, long)}
1032 * passing {@code this} as the argument. In this case, the unit determines
1033 * whether and how to perform the addition.
1034 * <p>
1035 * This instance is immutable and unaffected by this method call.
1036 *
1037 * @param amountToAdd the amount of the unit to add to the result, may be negative
1038 * @param unit the unit of the amount to add, not null
1039 * @return a {@code LocalTime} based on this time with the specified amount added, not null
1040 * @throws DateTimeException if the addition cannot be made
1041 * @throws UnsupportedOperationException if the unit is not supported
1042 * @throws ArithmeticException if numeric overflow occurs
1043 */
1044 @Override
1045 public LocalTime plus(long amountToAdd, TemporalUnit unit) {
1046     if (unit instanceof ChronoUnit) {
1047         switch ((ChronoUnit) unit) {
1048             case NANOS: return plusNanos(amountToAdd);
1049             case MICROS: return plusNanos((amountToAdd % MICROS_PER_DAY) * 1000);
1050             case MILLIS: return plusNanos((amountToAdd % MILLIS_PER_DAY) * 1000_000);
1051             case SECONDS: return plusSeconds(amountToAdd);
1052             case MINUTES: return plusMinutes(amountToAdd);
1053             case HOURS: return plusHours(amountToAdd);
1054             case HALF_DAYS: return plusHours((amountToAdd % 2) * 12);
1055         }
1056         throw new UnsupportedOperationException("Unsupported unit: " + unit);
1057     }
1058     return unit.addTo(this, amountToAdd);
1059 }

```

```

1059
1060 //-----
1061 /**
1062  * Returns a copy of this {@code LocalTime} with the specified number of hours added.
1063  * <p>
1064  * This adds the specified number of hours to this time, returning a new time.
1065  * The calculation wraps around midnight.
1066  * <p>
1067  * This instance is immutable and unaffected by this method call.
1068  *
1069  * @param hoursToAdd the hours to add, may be negative
1070  * @return a {@code LocalTime} based on this time with the hours added, not null
1071  */
1072 public LocalTime plusHours(long hoursToAdd) {
1073     if (hoursToAdd == 0) {
1074         return this;
1075     }
1076     int newHour = ((int) (hoursToAdd % HOURS_PER_DAY) + hour + HOURS_PER_DAY) % HOURS_PER_DAY;
1077     return create(newHour, minute, second, nano);
1078 }
1079
1080 /**
1081  * Returns a copy of this {@code LocalTime} with the specified number of minutes added.
1082  * <p>
1083  * This adds the specified number of minutes to this time, returning a new time.
1084  * The calculation wraps around midnight.
1085  * <p>
1086  * This instance is immutable and unaffected by this method call.
1087  *
1088  * @param minutesToAdd the minutes to add, may be negative
1089  * @return a {@code LocalTime} based on this time with the minutes added, not null
1090  */
1091 public LocalTime plusMinutes(long minutesToAdd) {
1092     if (minutesToAdd == 0) {
1093         return this;
1094     }
1095     int mofd = hour * MINUTES_PER_HOUR + minute;
1096     int newMofd = ((int) (minutesToAdd % MINUTES_PER_DAY) + mofd + MINUTES_PER_DAY) %
        MINUTES_PER_DAY;
1097     if (mofd == newMofd) {
1098         return this;
1099     }
1100     int newHour = newMofd / MINUTES_PER_HOUR;
1101     int newMinute = newMofd % MINUTES_PER_HOUR;
1102     return create(newHour, newMinute, second, nano);
1103 }
1104
1105 /**
1106  * Returns a copy of this {@code LocalTime} with the specified number of seconds added.
1107  * <p>
1108  * This adds the specified number of seconds to this time, returning a new time.
1109  * The calculation wraps around midnight.
1110  * <p>

```

```

1111     * This instance is immutable and unaffected by this method call.
1112     *
1113     * @param secondstoAdd the seconds to add, may be negative
1114     * @return a {@code LocalTime} based on this time with the seconds added, not null
1115     */
1116     public LocalTime plusSeconds(long secondstoAdd) {
1117         if (secondstoAdd == 0) {
1118             return this;
1119         }
1120         int sofd = hour * SECONDS_PER_HOUR +
1121             minute * SECONDS_PER_MINUTE + second;
1122         int newSofd = ((int) (secondstoAdd % SECONDS_PER_DAY) + sofd + SECONDS_PER_DAY) %
1123             SECONDS_PER_DAY;
1124         if (sofd == newSofd) {
1125             return this;
1126         }
1127         int newHour = newSofd / SECONDS_PER_HOUR;
1128         int newMinute = (newSofd / SECONDS_PER_MINUTE) % MINUTES_PER_HOUR;
1129         int newSecond = newSofd % SECONDS_PER_MINUTE;
1130         return create(newHour, newMinute, newSecond, nano);
1131     }
1132
1133     /**
1134     * Returns a copy of this {@code LocalTime} with the specified number of nanoseconds added.
1135     * <p>
1136     * This adds the specified number of nanoseconds to this time, returning a new time.
1137     * The calculation wraps around midnight.
1138     * <p>
1139     * This instance is immutable and unaffected by this method call.
1140     *
1141     * @param nanosToAdd the nanos to add, may be negative
1142     * @return a {@code LocalTime} based on this time with the nanoseconds added, not null
1143     */
1144     public LocalTime plusNanos(long nanosToAdd) {
1145         if (nanosToAdd == 0) {
1146             return this;
1147         }
1148         long nofd = toNanoOfDay();
1149         long newNofd = ((nanosToAdd % NANOS_PER_DAY) + nofd + NANOS_PER_DAY) % NANOS_PER_DAY;
1150         if (nofd == newNofd) {
1151             return this;
1152         }
1153         int newHour = (int) (newNofd / NANOS_PER_HOUR);
1154         int newMinute = (int) ((newNofd / NANOS_PER_MINUTE) % MINUTES_PER_HOUR);
1155         int newSecond = (int) ((newNofd / NANOS_PER_SECOND) % SECONDS_PER_MINUTE);
1156         int newNano = (int) (newNofd % NANOS_PER_SECOND);
1157         return create(newHour, newMinute, newSecond, newNano);
1158     }
1159
1160     /**
1161     * Returns a copy of this time with the specified amount subtracted.
1162     * <p>

```

```

1163     * This returns a {@code LocalTime}, based on this one, with the specified amount subtracted.
1164     * The amount is typically {@link Duration} but may be any other type implementing
1165     * the {@link TemporalAmount} interface.
1166     * <p>
1167     * The calculation is delegated to the amount object by calling
1168     * {@link TemporalAmount#subtractFrom(Temporal)}. The amount implementation is free
1169     * to implement the subtraction in any way it wishes, however it typically
1170     * calls back to {@link #minus(long, TemporalUnit)}. Consult the documentation
1171     * of the amount implementation to determine if it can be successfully subtracted.
1172     * <p>
1173     * This instance is immutable and unaffected by this method call.
1174     *
1175     * @param amountToSubtract the amount to subtract, not null
1176     * @return a {@code LocalTime} based on this time with the subtraction made, not null
1177     * @throws DateTimeException if the subtraction cannot be made
1178     * @throws ArithmeticException if numeric overflow occurs
1179     */
1180     @Override
1181     public LocalTime minus(TemporalAmount amountToSubtract) {
1182         return (LocalTime) amountToSubtract.subtractFrom(this);
1183     }
1184
1185     /**
1186     * Returns a copy of this time with the specified amount subtracted.
1187     * <p>
1188     * This returns a {@code LocalTime}, based on this one, with the amount
1189     * in terms of the unit subtracted. If it is not possible to subtract the amount,
1190     * because the unit is not supported or for some other reason, an exception is thrown.
1191     * <p>
1192     * This method is equivalent to {@link #plus(long, TemporalUnit)} with the amount negated.
1193     * See that method for a full description of how addition, and thus subtraction, works.
1194     * <p>
1195     * This instance is immutable and unaffected by this method call.
1196     *
1197     * @param amountToSubtract the amount of the unit to subtract from the result, may be negative
1198     * @param unit the unit of the amount to subtract, not null
1199     * @return a {@code LocalTime} based on this time with the specified amount subtracted, not
1200     *         null
1201     * @throws DateTimeException if the subtraction cannot be made
1202     * @throws UnsupportedTemporalTypeException if the unit is not supported
1203     * @throws ArithmeticException if numeric overflow occurs
1204     */
1205     @Override
1206     public LocalTime minus(long amountToSubtract, TemporalUnit unit) {
1207         return (amountToSubtract == Long.MIN_VALUE ? plus(Long.MAX_VALUE, unit).plus(1, unit) :
1208             plus(-amountToSubtract, unit));
1209     }
1210
1211     /**
1212     * Returns a copy of this {@code LocalTime} with the specified number of hours subtracted.
1213     * <p>
1214     * This subtracts the specified number of hours from this time, returning a new time.

```

```
1214     * The calculation wraps around midnight.
1215     * <p>
1216     * This instance is immutable and unaffected by this method call.
1217     *
1218     * @param hoursToSubtract the hours to subtract, may be negative
1219     * @return a {@code LocalTime} based on this time with the hours subtracted, not null
1220     */
1221     public LocalTime minusHours(long hoursToSubtract) {
1222         return plusHours(-(hoursToSubtract % HOURS_PER_DAY));
1223     }
1224
1225     /**
1226     * Returns a copy of this {@code LocalTime} with the specified number of minutes subtracted.
1227     * <p>
1228     * This subtracts the specified number of minutes from this time, returning a new time.
1229     * The calculation wraps around midnight.
1230     * <p>
1231     * This instance is immutable and unaffected by this method call.
1232     *
1233     * @param minutesToSubtract the minutes to subtract, may be negative
1234     * @return a {@code LocalTime} based on this time with the minutes subtracted, not null
1235     */
1236     public LocalTime minusMinutes(long minutesToSubtract) {
1237         return plusMinutes(-(minutesToSubtract % MINUTES_PER_DAY));
1238     }
1239
1240     /**
1241     * Returns a copy of this {@code LocalTime} with the specified number of seconds subtracted.
1242     * <p>
1243     * This subtracts the specified number of seconds from this time, returning a new time.
1244     * The calculation wraps around midnight.
1245     * <p>
1246     * This instance is immutable and unaffected by this method call.
1247     *
1248     * @param secondsToSubtract the seconds to subtract, may be negative
1249     * @return a {@code LocalTime} based on this time with the seconds subtracted, not null
1250     */
1251     public LocalTime minusSeconds(long secondsToSubtract) {
1252         return plusSeconds(-(secondsToSubtract % SECONDS_PER_DAY));
1253     }
1254
1255     /**
1256     * Returns a copy of this {@code LocalTime} with the specified number of nanoseconds subtracted
1257     * .
1258     * <p>
1259     * This subtracts the specified number of nanoseconds from this time, returning a new time.
1260     * The calculation wraps around midnight.
1261     * <p>
1262     * This instance is immutable and unaffected by this method call.
1263     *
1264     * @param nanosToSubtract the nanos to subtract, may be negative
1265     * @return a {@code LocalTime} based on this time with the nanoseconds subtracted, not null
1266     */
```

```

1266 public LocalTime minusNanos(long nanosToSubtract) {
1267     return plusNanos(-(nanosToSubtract % NANOS_PER_DAY));
1268 }
1269
1270 //-----
1271 /**
1272  * Queries this time using the specified query.
1273  * <p>
1274  * This queries this time using the specified query strategy object.
1275  * The {@code TemporalQuery} object defines the logic to be used to
1276  * obtain the result. Read the documentation of the query to understand
1277  * what the result of this method will be.
1278  * <p>
1279  * The result of this method is obtained by invoking the
1280  * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
1281  * specified query passing {@code this} as the argument.
1282  *
1283  * @param <R> the type of the result
1284  * @param query the query to invoke, not null
1285  * @return the query result, null may be returned (defined by the query)
1286  * @throws DateTimeException if unable to query (defined by the query)
1287  * @throws ArithmeticException if numeric overflow occurs (defined by the query)
1288  */
1289 @SuppressWarnings("unchecked")
1290 @Override
1291 public <R> R query(TemporalQuery<R> query) {
1292     if (query == TemporalQueries.chronology() || query == TemporalQueries.zoneId() ||
1293         query == TemporalQueries.zone() || query == TemporalQueries.offset()) {
1294         return null;
1295     } else if (query == TemporalQueries.localTime()) {
1296         return (R) this;
1297     } else if (query == TemporalQueries.localDate()) {
1298         return null;
1299     } else if (query == TemporalQueries.precision()) {
1300         return (R) NANOS;
1301     }
1302     // inline TemporalAccessor.super.query(query) as an optimization
1303     // non-JDK classes are not permitted to make this optimization
1304     return query.queryFrom(this);
1305 }
1306
1307 /**
1308  * Adjusts the specified temporal object to have the same time as this object.
1309  * <p>
1310  * This returns a temporal object of the same observable type as the input
1311  * with the time changed to be the same as this.
1312  * <p>
1313  * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
1314  * passing {@link ChronoField#NANO_OF_DAY} as the field.
1315  * <p>
1316  * In most cases, it is clearer to reverse the calling pattern by using
1317  * {@link Temporal#with(TemporalAdjuster)}:
1318  * <pre>

```



```

1319 * // these two lines are equivalent, but the second approach is recommended
1320 * temporal = thisLocalTime.adjustInto(temporal);
1321 * temporal = temporal.with(thisLocalTime);
1322 * </pre>
1323 * <p>
1324 * This instance is immutable and unaffected by this method call.
1325 *
1326 * @param temporal the target object to be adjusted, not null
1327 * @return the adjusted object, not null
1328 * @throws DateTimeException if unable to make the adjustment
1329 * @throws ArithmeticException if numeric overflow occurs
1330 */
1331 @Override
1332 public Temporal adjustInto(Temporal temporal) {
1333     return temporal.with(NANO_OF_DAY, toNanoOfDay());
1334 }
1335
1336 /**
1337 * Calculates the amount of time until another time in terms of the specified unit.
1338 * <p>
1339 * This calculates the amount of time between two {@code LocalTime}
1340 * objects in terms of a single {@code TemporalUnit}.
1341 * The start and end points are {@code this} and the specified time.
1342 * The result will be negative if the end is before the start.
1343 * The {@code Temporal} passed to this method is converted to a
1344 * {@code LocalTime} using {@link #from(TemporalAccessor)}.
1345 * For example, the amount in hours between two times can be calculated
1346 * using {@code startTime.until(endTime, HOURS)}.
1347 * <p>
1348 * The calculation returns a whole number, representing the number of
1349 * complete units between the two times.
1350 * For example, the amount in hours between 11:30 and 13:29 will only
1351 * be one hour as it is one minute short of two hours.
1352 * <p>
1353 * There are two equivalent ways of using this method.
1354 * The first is to invoke this method.
1355 * The second is to use {@link TemporalUnit#between(Temporal, Temporal)}:
1356 * <pre>
1357 * // these two lines are equivalent
1358 * amount = start.until(end, MINUTES);
1359 * amount = MINUTES.between(start, end);
1360 * </pre>
1361 * The choice should be made based on which makes the code more readable.
1362 * <p>
1363 * The calculation is implemented in this method for {@link ChronoUnit}.
1364 * The units {@code NANOS}, {@code MICROS}, {@code MILLIS}, {@code SECONDS},
1365 * {@code MINUTES}, {@code HOURS} and {@code HALF_DAYS} are supported.
1366 * Other {@code ChronoUnit} values will throw an exception.
1367 * <p>
1368 * If the unit is not a {@code ChronoUnit}, then the result of this method
1369 * is obtained by invoking {@code TemporalUnit.between(Temporal, Temporal)}
1370 * passing {@code this} as the first argument and the converted input temporal
1371 * as the second argument.

```



```

1372 * <p>
1373 * This instance is immutable and unaffected by this method call.
1374 *
1375 * @param endExclusive the end time, exclusive, which is converted to a {@code LocalTime}, not
    null
1376 * @param unit the unit to measure the amount in, not null
1377 * @return the amount of time between this time and the end time
1378 * @throws DateTimeException if the amount cannot be calculated, or the end
1379 * temporal cannot be converted to a {@code LocalTime}
1380 * @throws UnsupportedTemporalTypeException if the unit is not supported
1381 * @throws ArithmeticException if numeric overflow occurs
1382 */
1383 @Override
1384 public long until(Temporal endExclusive, TemporalUnit unit) {
1385     LocalTime end = LocalTime.from(endExclusive);
1386     if (unit instanceof ChronoUnit) {
1387         long nanosUntil = end.toNanoOfDay() - toNanoOfDay(); // no overflow
1388         switch ((ChronoUnit) unit) {
1389             case NANOS: return nanosUntil;
1390             case MICROS: return nanosUntil / 1000;
1391             case MILLIS: return nanosUntil / 1000_000;
1392             case SECONDS: return nanosUntil / NANOS_PER_SECOND;
1393             case MINUTES: return nanosUntil / NANOS_PER_MINUTE;
1394             case HOURS: return nanosUntil / NANOS_PER_HOUR;
1395             case HALF_DAYS: return nanosUntil / (12 * NANOS_PER_HOUR);
1396         }
1397         throw new UnsupportedTemporalTypeException("Unsupported unit: " + unit);
1398     }
1399     return unit.between(this, end);
1400 }
1401
1402 /**
1403 * Formats this time using the specified formatter.
1404 * <p>
1405 * This time will be passed to the formatter to produce a string.
1406 *
1407 * @param formatter the formatter to use, not null
1408 * @return the formatted time string, not null
1409 * @throws DateTimeException if an error occurs during printing
1410 */
1411 public String format(DateTimeFormatter formatter) {
1412     Objects.requireNonNull(formatter, "formatter");
1413     return formatter.format(this);
1414 }
1415
1416 //-----
1417 /**
1418 * Combines this time with a date to create a {@code LocalDateTime}.
1419 * <p>
1420 * This returns a {@code LocalDateTime} formed from this time at the specified date.
1421 * All possible combinations of date and time are valid.
1422 *
1423 * @param date the date to combine with, not null

```

```

1424     * @return the local date-time formed from this time and the specified date, not null
1425     */
1426     public LocalDateTime atDate(LocalDate date) {
1427         return LocalDateTime.of(date, this);
1428     }
1429
1430     /**
1431     * Combines this time with an offset to create an {@code OffsetTime}.
1432     * <p>
1433     * This returns an {@code OffsetTime} formed from this time at the specified offset.
1434     * All possible combinations of time and offset are valid.
1435     *
1436     * @param offset the offset to combine with, not null
1437     * @return the offset time formed from this time and the specified offset, not null
1438     */
1439     public OffsetTime atOffset(ZoneOffset offset) {
1440         return OffsetTime.of(this, offset);
1441     }
1442
1443     //-----
1444     /**
1445     * Extracts the time as seconds of day,
1446     * from {@code 0} to {@code 24 * 60 * 60 - 1}.
1447     *
1448     * @return the second-of-day equivalent to this time
1449     */
1450     public int toSecondOfDay() {
1451         int total = hour * SECONDS_PER_HOUR;
1452         total += minute * SECONDS_PER_MINUTE;
1453         total += second;
1454         return total;
1455     }
1456
1457     /**
1458     * Extracts the time as nanos of day,
1459     * from {@code 0} to {@code 24 * 60 * 60 * 1,000,000,000 - 1}.
1460     *
1461     * @return the nano of day equivalent to this time
1462     */
1463     public long toNanoOfDay() {
1464         long total = hour * NANOS_PER_HOUR;
1465         total += minute * NANOS_PER_MINUTE;
1466         total += second * NANOS_PER_SECOND;
1467         total += nano;
1468         return total;
1469     }
1470
1471     //-----
1472     /**
1473     * Compares this time to another time.
1474     * <p>
1475     * The comparison is based on the time-line position of the local times within a day.
1476     * It is "consistent with equals", as defined by {@link Comparable}.

```

```

1477     *
1478     * @param other the other time to compare to, not null
1479     * @return the comparator value, negative if less, positive if greater
1480     * @throws NullPointerException if {@code other} is null
1481     */
1482     @Override
1483     public int compareTo(LocalTime other) {
1484         int cmp = Integer.compare(hour, other.hour);
1485         if (cmp == 0) {
1486             cmp = Integer.compare(minute, other.minute);
1487             if (cmp == 0) {
1488                 cmp = Integer.compare(second, other.second);
1489                 if (cmp == 0) {
1490                     cmp = Integer.compare(nano, other.nano);
1491                 }
1492             }
1493         }
1494         return cmp;
1495     }
1496
1497     /**
1498     * Checks if this time is after the specified time.
1499     * <p>
1500     * The comparison is based on the time-line position of the time within a day.
1501     *
1502     * @param other the other time to compare to, not null
1503     * @return true if this is after the specified time
1504     * @throws NullPointerException if {@code other} is null
1505     */
1506     public boolean isAfter(LocalTime other) {
1507         return compareTo(other) > 0;
1508     }
1509
1510     /**
1511     * Checks if this time is before the specified time.
1512     * <p>
1513     * The comparison is based on the time-line position of the time within a day.
1514     *
1515     * @param other the other time to compare to, not null
1516     * @return true if this point is before the specified time
1517     * @throws NullPointerException if {@code other} is null
1518     */
1519     public boolean isBefore(LocalTime other) {
1520         return compareTo(other) < 0;
1521     }
1522
1523     //-----
1524     /**
1525     * Checks if this time is equal to another time.
1526     * <p>
1527     * The comparison is based on the time-line position of the time within a day.
1528     * <p>
1529     * Only objects of type {@code LocalDateTime} are compared, other types return false.

```

```

1530     * To compare the date of two {@code TemporalAccessor} instances, use
1531     * {@link ChronoField#NANO_OF_DAY} as a comparator.
1532     *
1533     * @param obj the object to check, null returns false
1534     * @return true if this is equal to the other time
1535     */
1536     @Override
1537     public boolean equals(Object obj) {
1538         if (this == obj) {
1539             return true;
1540         }
1541         if (obj instanceof LocalDateTime) {
1542             LocalDateTime other = (LocalDateTime) obj;
1543             return hour == other.hour && minute == other.minute &&
1544                 second == other.second && nano == other.nano;
1545         }
1546         return false;
1547     }
1548
1549     /**
1550     * A hash code for this time.
1551     *
1552     * @return a suitable hash code
1553     */
1554     @Override
1555     public int hashCode() {
1556         long nod = toNanoOfDay();
1557         return (int) (nod ^ (nod >>> 32));
1558     }
1559
1560     //-----
1561     /**
1562     * Outputs this time as a {@code String}, such as {@code 10:15}.
1563     * <p>
1564     * The output will be one of the following ISO-8601 formats:
1565     * <ul>
1566     * <li>{@code HH:mm}</li>
1567     * <li>{@code HH:mm:ss}</li>
1568     * <li>{@code HH:mm:ss.SSS}</li>
1569     * <li>{@code HH:mm:ss.SSSSSS}</li>
1570     * <li>{@code HH:mm:ss.SSSSSSSS}</li>
1571     * </ul>
1572     * The format used will be the shortest that outputs the full value of
1573     * the time where the omitted parts are implied to be zero.
1574     *
1575     * @return a string representation of this time, not null
1576     */
1577     @Override
1578     public String toString() {
1579         StringBuilder buf = new StringBuilder(18);
1580         int hourValue = hour;
1581         int minuteValue = minute;
1582         int secondValue = second;

```

```

1583     int nanoValue = nano;
1584     buf.append(hourValue < 10 ? "0" : "").append(hourValue)
1585         .append(minuteValue < 10 ? ":0" : ":").append(minuteValue);
1586     if (secondValue > 0 || nanoValue > 0) {
1587         buf.append(secondValue < 10 ? ":0" : ":").append(secondValue);
1588         if (nanoValue > 0) {
1589             buf.append('.');
1590             if (nanoValue % 1000_000 == 0) {
1591                 buf.append(Integer.toString((nanoValue / 1000_000) + 1000).substring(1));
1592             } else if (nanoValue % 1000 == 0) {
1593                 buf.append(Integer.toString((nanoValue / 1000) + 1000_000).substring(1));
1594             } else {
1595                 buf.append(Integer.toString((nanoValue) + 1000_000_000).substring(1));
1596             }
1597         }
1598     }
1599     return buf.toString();
1600 }
1601
1602 //-----
1603 /**
1604  * Writes the object using a
1605  * <a href="../../serialized-form.html#java.time.Ser">dedicated serialized form</a>.
1606  * @serialData
1607  * A twos-complement value indicates the remaining values are not in the stream
1608  * and should be set to zero.
1609  * <pre>
1610  * out.writeByte(4); // identifies a LocalTime
1611  * if (nano == 0) {
1612  *     if (second == 0) {
1613  *         if (minute == 0) {
1614  *             out.writeByte(~hour);
1615  *         } else {
1616  *             out.writeByte(hour);
1617  *             out.writeByte(~minute);
1618  *         }
1619  *     } else {
1620  *         out.writeByte(hour);
1621  *         out.writeByte(minute);
1622  *         out.writeByte(~second);
1623  *     }
1624  * } else {
1625  *     out.writeByte(hour);
1626  *     out.writeByte(minute);
1627  *     out.writeByte(second);
1628  *     out.writeInt(nano);
1629  * }
1630  * </pre>
1631  *
1632  * @return the instance of {@code Ser}, not null
1633  */
1634 private Object writeReplace() {
1635     return new Ser(Ser.LOCAL_TIME_TYPE, this);

```

```
1636     }
1637
1638     /**
1639      * Defend against malicious streams.
1640      *
1641      * @param s the stream to read
1642      * @throws InvalidObjectException always
1643      */
1644     private void readObject(ObjectInputStream s) throws InvalidObjectException {
1645         throw new InvalidObjectException("Deserialization via serialization delegate");
1646     }
1647
1648     void writeExternal(DataOutput out) throws IOException {
1649         if (nano == 0) {
1650             if (second == 0) {
1651                 if (minute == 0) {
1652                     out.writeByte(~hour);
1653                 } else {
1654                     out.writeByte(hour);
1655                     out.writeByte(~minute);
1656                 }
1657             } else {
1658                 out.writeByte(hour);
1659                 out.writeByte(minute);
1660                 out.writeByte(~second);
1661             }
1662         } else {
1663             out.writeByte(hour);
1664             out.writeByte(minute);
1665             out.writeByte(second);
1666             out.writeInt(nano);
1667         }
1668     }
1669
1670     static LocalTime readExternal(DataInput in) throws IOException {
1671         int hour = in.readByte();
1672         int minute = 0;
1673         int second = 0;
1674         int nano = 0;
1675         if (hour < 0) {
1676             hour = ~hour;
1677         } else {
1678             minute = in.readByte();
1679             if (minute < 0) {
1680                 minute = ~minute;
1681             } else {
1682                 second = in.readByte();
1683                 if (second < 0) {
1684                     second = ~second;
1685                 } else {
1686                     nano = in.readInt();
1687                 }
1688             }
1689         }
1690     }
```

```
1689     }
1690     return LocalTime.of(hour, minute, second, nano);
1691 }
1692
1693 }
```

4.9 Month.java

Secuencia 58: java/time/Month.java

```
1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
```

```
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time;
63
64  import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
65  import static java.time.temporal.ChronoUnit.MONTHS;
66
67  import java.time.chrono.Chronology;
68  import java.time.chrono.IsoChronology;
69  import java.time.format.DateTimeFormatterBuilder;
70  import java.time.format.TextStyle;
71  import java.time.temporal.ChronoField;
72  import java.time.temporal.Temporal;
73  import java.time.temporal.TemporalAccessor;
74  import java.time.temporal.TemporalAdjuster;
75  import java.time.temporal.TemporalField;
76  import java.time.temporal.TemporalQueries;
77  import java.time.temporal.TemporalQuery;
78  import java.time.temporal.UnsupportedTemporalTypeException;
79  import java.time.temporal.ValueRange;
80  import java.util.Locale;
81
82  /**
83   * A month-of-year, such as 'July'.
84   * <p>
85   * {@code Month} is an enum representing the 12 months of the year -
86   * January, February, March, April, May, June, July, August, September, October,
87   * November and December.
88   * <p>
89   * In addition to the textual enum name, each month-of-year has an {@code int} value.
90   * The {@code int} value follows normal usage and the ISO-8601 standard,
91   * from 1 (January) to 12 (December). It is recommended that applications use the enum
92   * rather than the {@code int} value to ensure code clarity.
93   * <p>
94   * <b>Do not use {@code ordinal()} to obtain the numeric representation of {@code Month}.
95   * Use {@code getValue()} instead.</b>
96   * <p>
97   * This enum represents a common concept that is found in many calendar systems.
```



```
98  * As such, this enum may be used by any calendar system that has the month-of-year
99  * concept defined exactly equivalent to the ISO-8601 calendar system.
100 *
101 * @implSpec
102 * This is an immutable and thread-safe enum.
103 *
104 * @since 1.8
105 */
106 public enum Month implements TemporalAccessor, TemporalAdjuster {
107
108     /**
109      * The singleton instance for the month of January with 31 days.
110      * This has the numeric value of {@code 1}.
111      */
112     JANUARY,
113     /**
114      * The singleton instance for the month of February with 28 days, or 29 in a leap year.
115      * This has the numeric value of {@code 2}.
116      */
117     FEBRUARY,
118     /**
119      * The singleton instance for the month of March with 31 days.
120      * This has the numeric value of {@code 3}.
121      */
122     MARCH,
123     /**
124      * The singleton instance for the month of April with 30 days.
125      * This has the numeric value of {@code 4}.
126      */
127     APRIL,
128     /**
129      * The singleton instance for the month of May with 31 days.
130      * This has the numeric value of {@code 5}.
131      */
132     MAY,
133     /**
134      * The singleton instance for the month of June with 30 days.
135      * This has the numeric value of {@code 6}.
136      */
137     JUNE,
138     /**
139      * The singleton instance for the month of July with 31 days.
140      * This has the numeric value of {@code 7}.
141      */
142     JULY,
143     /**
144      * The singleton instance for the month of August with 31 days.
145      * This has the numeric value of {@code 8}.
146      */
147     AUGUST,
148     /**
149      * The singleton instance for the month of September with 30 days.
150      * This has the numeric value of {@code 9}.
```

```

151     */
152     SEPTEMBER,
153     /**
154      * The singleton instance for the month of October with 31 days.
155      * This has the numeric value of {@code 10}.
156      */
157     OCTOBER,
158     /**
159      * The singleton instance for the month of November with 30 days.
160      * This has the numeric value of {@code 11}.
161      */
162     NOVEMBER,
163     /**
164      * The singleton instance for the month of December with 31 days.
165      * This has the numeric value of {@code 12}.
166      */
167     DECEMBER;
168     /**
169      * Private cache of all the constants.
170      */
171     private static final Month[] ENUMS = Month.values();
172
173     //-----
174     /**
175      * Obtains an instance of {@code Month} from an {@code int} value.
176      * <p>
177      * {@code Month} is an enum representing the 12 months of the year.
178      * This factory allows the enum to be obtained from the {@code int} value.
179      * The {@code int} value follows the ISO-8601 standard, from 1 (January) to 12 (December).
180      *
181      * @param month the month-of-year to represent, from 1 (January) to 12 (December)
182      * @return the month-of-year, not null
183      * @throws DateTimeException if the month-of-year is invalid
184      */
185     public static Month of(int month) {
186         if (month < 1 || month > 12) {
187             throw new DateTimeException("Invalid value for MonthOfYear: " + month);
188         }
189         return ENUMS[month - 1];
190     }
191
192     //-----
193     /**
194      * Obtains an instance of {@code Month} from a temporal object.
195      * <p>
196      * This obtains a month based on the specified temporal.
197      * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
198      * which this factory converts to an instance of {@code Month}.
199      * <p>
200      * The conversion extracts the {@link ChronoField#MONTH_OF_YEAR MONTH_OF_YEAR} field.
201      * The extraction is only permitted if the temporal object has an ISO
202      * chronology, or can be converted to a {@code LocalDate}.
203      * <p>

```

```

204 * This method matches the signature of the functional interface {@link TemporalQuery}
205 * allowing it to be used as a query via method reference, {@code Month::from}.
206 *
207 * @param temporal the temporal object to convert, not null
208 * @return the month-of-year, not null
209 * @throws DateTimeException if unable to convert to a {@code Month}
210 */
211 public static Month from(TemporalAccessor temporal) {
212     if (temporal instanceof Month) {
213         return (Month) temporal;
214     }
215     try {
216         if (IsoChronology.INSTANCE.equals(Chronology.from(temporal)) == false) {
217             temporal = LocalDate.from(temporal);
218         }
219         return of(temporal.get(MONTH_OF_YEAR));
220     } catch (DateTimeException ex) {
221         throw new DateTimeException("Unable to obtain Month from TemporalAccessor: " +
222             temporal + " of type " + temporal.getClass().getName(), ex);
223     }
224 }
225
226 //-----
227 /**
228  * Gets the month-of-year {@code int} value.
229  * <p>
230  * The values are numbered following the ISO-8601 standard,
231  * from 1 (January) to 12 (December).
232  *
233  * @return the month-of-year, from 1 (January) to 12 (December)
234  */
235 public int getValue() {
236     return ordinal() + 1;
237 }
238
239 //-----
240 /**
241  * Gets the textual representation, such as 'Jan' or 'December'.
242  * <p>
243  * This returns the textual name used to identify the month-of-year,
244  * suitable for presentation to the user.
245  * The parameters control the style of the returned text and the locale.
246  * <p>
247  * If no textual mapping is found then the {@link #getValue() numeric value} is returned.
248  *
249  * @param style the length of the text required, not null
250  * @param locale the locale to use, not null
251  * @return the text value of the month-of-year, not null
252  */
253 public String getDisplayName(TextStyle style, Locale locale) {
254     return new DateTimeFormatterBuilder().appendText(MONTH_OF_YEAR, style).toFormatter(locale).
255         format(this);
256 }

```

```

256
257 //-----
258 /**
259  * Checks if the specified field is supported.
260  * <p>
261  * This checks if this month-of-year can be queried for the specified field.
262  * If false, then calling the {@link #range(TemporalField) range} and
263  * {@link #get(TemporalField) get} methods will throw an exception.
264  * <p>
265  * If the field is {@link ChronoField#MONTH_OF_YEAR MONTH_OF_YEAR} then
266  * this method returns true.
267  * All other {@code ChronoField} instances will return false.
268  * <p>
269  * If the field is not a {@code ChronoField}, then the result of this method
270  * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
271  * passing {@code this} as the argument.
272  * Whether the field is supported is determined by the field.
273  *
274  * @param field the field to check, null returns false
275  * @return true if the field is supported on this month-of-year, false if not
276  */
277 @Override
278 public boolean isSupported(TemporalField field) {
279     if (field instanceof ChronoField) {
280         return field == MONTH_OF_YEAR;
281     }
282     return field != null && field.isSupportedBy(this);
283 }
284
285 /**
286  * Gets the range of valid values for the specified field.
287  * <p>
288  * The range object expresses the minimum and maximum valid values for a field.
289  * This month is used to enhance the accuracy of the returned range.
290  * If it is not possible to return the range, because the field is not supported
291  * or for some other reason, an exception is thrown.
292  * <p>
293  * If the field is {@link ChronoField#MONTH_OF_YEAR MONTH_OF_YEAR} then the
294  * range of the month-of-year, from 1 to 12, will be returned.
295  * All other {@code ChronoField} instances will throw an {@code
296     UnsupportedTemporalTypeException}.
297  * <p>
298  * If the field is not a {@code ChronoField}, then the result of this method
299  * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
300  * passing {@code this} as the argument.
301  * Whether the range can be obtained is determined by the field.
302  *
303  * @param field the field to query the range for, not null
304  * @return the range of valid values for the field, not null
305  * @throws DateTimeException if the range for the field cannot be obtained
306  * @throws UnsupportedTemporalTypeException if the field is not supported
307  */
308 @Override

```

```

308 public ValueRange range(TemporalField field) {
309     if (field == MONTH_OF_YEAR) {
310         return field.range();
311     }
312     return TemporalAccessor.super.range(field);
313 }
314
315 /**
316  * Gets the value of the specified field from this month-of-year as an {@code int}.
317  * <p>
318  * This queries this month for the value of the specified field.
319  * The returned value will always be within the valid range of values for the field.
320  * If it is not possible to return the value, because the field is not supported
321  * or for some other reason, an exception is thrown.
322  * <p>
323  * If the field is {@link ChronoField#MONTH_OF_YEAR MONTH_OF_YEAR} then the
324  * value of the month-of-year, from 1 to 12, will be returned.
325  * All other {@code ChronoField} instances will throw an {@code
326     * UnsupportedTemporalTypeException}.
327  * <p>
328  * If the field is not a {@code ChronoField}, then the result of this method
329  * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
330  * passing {@code this} as the argument. Whether the value can be obtained,
331  * and what the value represents, is determined by the field.
332  *
333  * @param field the field to get, not null
334  * @return the value for the field, within the valid range of values
335  * @throws DateTimeException if a value for the field cannot be obtained or
336  *         the value is outside the range of valid values for the field
337  * @throws UnsupportedTemporalTypeException if the field is not supported or
338  *         the range of values exceeds an {@code int}
339  * @throws ArithmeticException if numeric overflow occurs
340  */
341 @Override
342 public int get(TemporalField field) {
343     if (field == MONTH_OF_YEAR) {
344         return getValue();
345     }
346     return TemporalAccessor.super.get(field);
347 }
348
349 /**
350  * Gets the value of the specified field from this month-of-year as a {@code long}.
351  * <p>
352  * This queries this month for the value of the specified field.
353  * If it is not possible to return the value, because the field is not supported
354  * or for some other reason, an exception is thrown.
355  * <p>
356  * If the field is {@link ChronoField#MONTH_OF_YEAR MONTH_OF_YEAR} then the
357  * value of the month-of-year, from 1 to 12, will be returned.
358  * All other {@code ChronoField} instances will throw an {@code

```

```

359     * If the field is not a {@code ChronoField}, then the result of this method
360     * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
361     * passing {@code this} as the argument. Whether the value can be obtained,
362     * and what the value represents, is determined by the field.
363     *
364     * @param field the field to get, not null
365     * @return the value for the field
366     * @throws DateTimeException if a value for the field cannot be obtained
367     * @throws UnsupportedTemporalTypeException if the field is not supported
368     * @throws ArithmeticException if numeric overflow occurs
369     */
370     @Override
371     public long getLong(TemporalField field) {
372         if (field == MONTH_OF_YEAR) {
373             return getValue();
374         } else if (field instanceof ChronoField) {
375             throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
376         }
377         return field.getFrom(this);
378     }
379
380     //-----
381     /**
382     * Returns the month-of-year that is the specified number of quarters after this one.
383     * <p>
384     * The calculation rolls around the end of the year from December to January.
385     * The specified period may be negative.
386     * <p>
387     * This instance is immutable and unaffected by this method call.
388     *
389     * @param months the months to add, positive or negative
390     * @return the resulting month, not null
391     */
392     public Month plus(long months) {
393         int amount = (int) (months % 12);
394         return ENUMS[(ordinal() + (amount + 12)) % 12];
395     }
396
397     /**
398     * Returns the month-of-year that is the specified number of months before this one.
399     * <p>
400     * The calculation rolls around the start of the year from January to December.
401     * The specified period may be negative.
402     * <p>
403     * This instance is immutable and unaffected by this method call.
404     *
405     * @param months the months to subtract, positive or negative
406     * @return the resulting month, not null
407     */
408     public Month minus(long months) {
409         return plus(-(months % 12));
410     }
411

```

```
412 //-----
413 /**
414  * Gets the length of this month in days.
415  * <p>
416  * This takes a flag to determine whether to return the length for a leap year or not.
417  * <p>
418  * February has 28 days in a standard year and 29 days in a leap year.
419  * April, June, September and November have 30 days.
420  * All other months have 31 days.
421  *
422  * @param leapYear true if the length is required for a leap year
423  * @return the length of this month in days, from 28 to 31
424  */
425 public int length(boolean leapYear) {
426     switch (this) {
427         case FEBRUARY:
428             return (leapYear ? 29 : 28);
429         case APRIL:
430         case JUNE:
431         case SEPTEMBER:
432         case NOVEMBER:
433             return 30;
434         default:
435             return 31;
436     }
437 }
438
439 /**
440  * Gets the minimum length of this month in days.
441  * <p>
442  * February has a minimum length of 28 days.
443  * April, June, September and November have 30 days.
444  * All other months have 31 days.
445  *
446  * @return the minimum length of this month in days, from 28 to 31
447  */
448 public int minLength() {
449     switch (this) {
450         case FEBRUARY:
451             return 28;
452         case APRIL:
453         case JUNE:
454         case SEPTEMBER:
455         case NOVEMBER:
456             return 30;
457         default:
458             return 31;
459     }
460 }
461
462 /**
463  * Gets the maximum length of this month in days.
464  * <p>
```

```
465     * February has a maximum length of 29 days.
466     * April, June, September and November have 30 days.
467     * All other months have 31 days.
468     *
469     * @return the maximum length of this month in days, from 29 to 31
470     */
471     public int maxLength() {
472         switch (this) {
473             case FEBRUARY:
474                 return 29;
475             case APRIL:
476             case JUNE:
477             case SEPTEMBER:
478             case NOVEMBER:
479                 return 30;
480             default:
481                 return 31;
482         }
483     }
484
485     //-----
486     /**
487     * Gets the day-of-year corresponding to the first day of this month.
488     * <p>
489     * This returns the day-of-year that this month begins on, using the leap
490     * year flag to determine the length of February.
491     *
492     * @param leapYear true if the length is required for a leap year
493     * @return the day of year corresponding to the first day of this month, from 1 to 336
494     */
495     public int firstDayOfYear(boolean leapYear) {
496         int leap = leapYear ? 1 : 0;
497         switch (this) {
498             case JANUARY:
499                 return 1;
500             case FEBRUARY:
501                 return 32;
502             case MARCH:
503                 return 60 + leap;
504             case APRIL:
505                 return 91 + leap;
506             case MAY:
507                 return 121 + leap;
508             case JUNE:
509                 return 152 + leap;
510             case JULY:
511                 return 182 + leap;
512             case AUGUST:
513                 return 213 + leap;
514             case SEPTEMBER:
515                 return 244 + leap;
516             case OCTOBER:
517                 return 274 + leap;
```



```

518         case NOVEMBER:
519             return 305 + leap;
520         case DECEMBER:
521             default:
522                 return 335 + leap;
523     }
524 }
525
526 /**
527  * Gets the month corresponding to the first month of this quarter.
528  * <p>
529  * The year can be divided into four quarters.
530  * This method returns the first month of the quarter for the base month.
531  * January, February and March return January.
532  * April, May and June return April.
533  * July, August and September return July.
534  * October, November and December return October.
535  *
536  * @return the first month of the quarter corresponding to this month, not null
537  */
538 public Month firstMonthOfQuarter() {
539     return ENUMS[(ordinal() / 3) * 3];
540 }
541
542 //-----
543 /**
544  * Queries this month-of-year using the specified query.
545  * <p>
546  * This queries this month-of-year using the specified query strategy object.
547  * The {@code TemporalQuery} object defines the logic to be used to
548  * obtain the result. Read the documentation of the query to understand
549  * what the result of this method will be.
550  * <p>
551  * The result of this method is obtained by invoking the
552  * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
553  * specified query passing {@code this} as the argument.
554  *
555  * @param <R> the type of the result
556  * @param query the query to invoke, not null
557  * @return the query result, null may be returned (defined by the query)
558  * @throws DateTimeException if unable to query (defined by the query)
559  * @throws ArithmeticException if numeric overflow occurs (defined by the query)
560  */
561 @SuppressWarnings("unchecked")
562 @Override
563 public <R> query(TemporalQuery<R> query) {
564     if (query == TemporalQueries.chronology()) {
565         return (R) IsoChronology.INSTANCE;
566     } else if (query == TemporalQueries.precision()) {
567         return (R) MONTHS;
568     }
569     return TemporalAccessor.super.query(query);
570 }

```

```

571
572 /**
573  * Adjusts the specified temporal object to have this month-of-year.
574  * <p>
575  * This returns a temporal object of the same observable type as the input
576  * with the month-of-year changed to be the same as this.
577  * <p>
578  * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
579  * passing {@link ChronoField#MONTH_OF_YEAR} as the field.
580  * If the specified temporal object does not use the ISO calendar system then
581  * a {@code DateTimeException} is thrown.
582  * <p>
583  * In most cases, it is clearer to reverse the calling pattern by using
584  * {@link Temporal#with(TemporalAdjuster)}:
585  * <pre>
586  * // these two lines are equivalent, but the second approach is recommended
587  * temporal = thisMonth.adjustInto(temporal);
588  * temporal = temporal.with(thisMonth);
589  * </pre>
590  * <p>
591  * For example, given a date in May, the following are output:
592  * <pre>
593  * dateInMay.with(JANUARY);    // four months earlier
594  * dateInMay.with(APRIL);     // one months earlier
595  * dateInMay.with(MAY);       // same date
596  * dateInMay.with(JUNE);      // one month later
597  * dateInMay.with(DECEMBER);  // seven months later
598  * </pre>
599  * <p>
600  * This instance is immutable and unaffected by this method call.
601  *
602  * @param temporal the target object to be adjusted, not null
603  * @return the adjusted object, not null
604  * @throws DateTimeException if unable to make the adjustment
605  * @throws ArithmeticException if numeric overflow occurs
606  */
607 @Override
608 public Temporal adjustInto(Temporal temporal) {
609     if (Chronology.from(temporal).equals(IsoChronology.INSTANCE) == false) {
610         throw new DateTimeException("Adjustment only supported on ISO date-time");
611     }
612     return temporal.with(MONTH_OF_YEAR, getValue());
613 }
614
615 }

```

4.10 MonthDay.java

Secuencia 59: java/time/MonthDay.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *

```

```
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
```

```
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time;
63
64  import static java.time.temporal.ChronoField.DAY_OF_MONTH;
65  import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
66
67  import java.io.DataInput;
68  import java.io.DataOutput;
69  import java.io.IOException;
70  import java.io.InvalidObjectException;
71  import java.io.ObjectInputStream;
72  import java.io.Serializable;
73  import java.time.chrono.Chronology;
74  import java.time.chrono.IsoChronology;
75  import java.time.format.DateTimeFormatter;
76  import java.time.format.DateTimeFormatterBuilder;
77  import java.time.format.DateTimeParseException;
78  import java.time.temporal.ChronoField;
79  import java.time.temporal.Temporal;
80  import java.time.temporal.TemporalAccessor;
81  import java.time.temporal.TemporalAdjuster;
82  import java.time.temporal.TemporalField;
83  import java.time.temporal.TemporalQueries;
84  import java.time.temporal.TemporalQuery;
85  import java.time.temporal.UnsupportedTemporalTypeException;
86  import java.time.temporal.ValueRange;
87  import java.util.Objects;
88
89  /**
90   * A month-day in the ISO-8601 calendar system, such as {@code --12-03}.
91   * <p>
92   * {@code MonthDay} is an immutable date-time object that represents the combination
93   * of a month and day-of-month. Any field that can be derived from a month and day,
94   * such as quarter-of-year, can be obtained.
95   * <p>
96   * This class does not store or represent a year, time or time-zone.
97   * For example, the value "December 3rd" can be stored in a {@code MonthDay}.
98   * <p>
99   * Since a {@code MonthDay} does not possess a year, the leap day of
100  * February 29th is considered valid.
101  * <p>
102  * This class implements {@link TemporalAccessor} rather than {@link Temporal}.
103  * This is because it is not possible to define whether February 29th is valid or not
104  * without external information, preventing the implementation of plus/minus.
105  * Related to this, {@code MonthDay} only provides access to query and set the fields
106  * {@code MONTH_OF_YEAR} and {@code DAY_OF_MONTH}.
107  * <p>
108  * The ISO-8601 calendar system is the modern civil calendar system used today
109  * in most of the world. It is equivalent to the proleptic Gregorian calendar
110  * system, in which today's rules for leap years are applied for all time.
```

```

111 * For most applications written today, the ISO-8601 rules are entirely suitable.
112 * However, any application that makes use of historical dates, and requires them
113 * to be accurate will find the ISO-8601 approach unsuitable.
114 *
115 * <p>
116 * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
117 * class; use of identity-sensitive operations (including reference equality
118 * ({@code ==}), identity hash code, or synchronization) on instances of
119 * {@code MonthDay} may have unpredictable results and should be avoided.
120 * The {@code equals} method should be used for comparisons.
121 *
122 * @implSpec
123 * This class is immutable and thread-safe.
124 *
125 * @since 1.8
126 */
127 public final class MonthDay
128     implements TemporalAccessor, TemporalAdjuster, Comparable<MonthDay>, Serializable {
129
130     /**
131      * Serialization version.
132      */
133     private static final long serialVersionUID = -939150713474957432L;
134
135     /**
136      * Parser.
137      */
138     private static final DateTimeFormatter PARSER = new DateTimeFormatterBuilder()
139         .appendLiteral("--")
140         .appendValue(MONTH_OF_YEAR, 2)
141         .appendLiteral('-')
142         .appendValue(DAY_OF_MONTH, 2)
143         .toFormatter();
144
145     /**
146      * The month-of-year, not null.
147      */
148     private final int month;
149
150     /**
151      * The day-of-month.
152      */
153     private final int day;
154
155     //-----
156     /**
157      * Obtains the current month-day from the system clock in the default time-zone.
158      * <p>
159      * This will query the {@link Clock#systemDefaultZone() system clock} in the default
160      * time-zone to obtain the current month-day.
161      * <p>
162      * Using this method will prevent the ability to use an alternate clock for testing
163      * because the clock is hard-coded.
164      *
165      * @return the current month-day using the system clock and default time-zone, not null

```

```

164     */
165     public static MonthDay now() {
166         return now(Clock.systemDefaultZone());
167     }
168
169     /**
170      * Obtains the current month-day from the system clock in the specified time-zone.
171      * <p>
172      * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current month-
173      * day.
174      * Specifying the time-zone avoids dependence on the default time-zone.
175      * <p>
176      * Using this method will prevent the ability to use an alternate clock for testing
177      * because the clock is hard-coded.
178      *
179      * @param zone the zone ID to use, not null
180      * @return the current month-day using the system clock, not null
181      */
182     public static MonthDay now(ZoneId zone) {
183         return now(Clock.system(zone));
184     }
185
186     /**
187      * Obtains the current month-day from the specified clock.
188      * <p>
189      * This will query the specified clock to obtain the current month-day.
190      * Using this method allows the use of an alternate clock for testing.
191      * The alternate clock may be introduced using {@link Clock dependency injection}.
192      *
193      * @param clock the clock to use, not null
194      * @return the current month-day, not null
195      */
196     public static MonthDay now(Clock clock) {
197         final LocalDate now = LocalDate.now(clock); // called once
198         return MonthDay.of(now.getMonth(), now.getDayOfMonth());
199     }
200
201     //-----
202     /**
203      * Obtains an instance of {@code MonthDay}.
204      * <p>
205      * The day-of-month must be valid for the month within a leap year.
206      * Hence, for February, day 29 is valid.
207      * <p>
208      * For example, passing in April and day 31 will throw an exception, as
209      * there can never be April 31st in any year. By contrast, passing in
210      * February 29th is permitted, as that month-day can sometimes be valid.
211      *
212      * @param month the month-of-year to represent, not null
213      * @param dayOfMonth the day-of-month to represent, from 1 to 31
214      * @return the month-day, not null
215      * @throws DateTimeException if the value of any field is out of range,
216      * or if the day-of-month is invalid for the month

```

```

216     */
217     public static MonthDay of(Month month, int dayOfMonth) {
218         Objects.requireNonNull(month, "month");
219         DAY_OF_MONTH.checkValidValue(dayOfMonth);
220         if (dayOfMonth > month.maxLength()) {
221             throw new DateTimeException("Illegal value for DayOfMonth field, value " + dayOfMonth +
222                 " is not valid for month " + month.name());
223         }
224         return new MonthDay(month.getValue(), dayOfMonth);
225     }
226
227     /**
228      * Obtains an instance of {@code MonthDay}.
229      * <p>
230      * The day-of-month must be valid for the month within a leap year.
231      * Hence, for month 2 (February), day 29 is valid.
232      * <p>
233      * For example, passing in month 4 (April) and day 31 will throw an exception, as
234      * there can never be April 31st in any year. By contrast, passing in
235      * February 29th is permitted, as that month-day can sometimes be valid.
236      *
237      * @param month the month-of-year to represent, from 1 (January) to 12 (December)
238      * @param dayOfMonth the day-of-month to represent, from 1 to 31
239      * @return the month-day, not null
240      * @throws DateTimeException if the value of any field is out of range,
241      * or if the day-of-month is invalid for the month
242      */
243     public static MonthDay of(int month, int dayOfMonth) {
244         return of(Month.of(month), dayOfMonth);
245     }
246
247     //-----
248     /**
249      * Obtains an instance of {@code MonthDay} from a temporal object.
250      * <p>
251      * This obtains a month-day based on the specified temporal.
252      * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
253      * which this factory converts to an instance of {@code MonthDay}.
254      * <p>
255      * The conversion extracts the {@link ChronoField#MONTH_OF_YEAR MONTH_OF_YEAR} and
256      * {@link ChronoField#DAY_OF_MONTH DAY_OF_MONTH} fields.
257      * The extraction is only permitted if the temporal object has an ISO
258      * chronology, or can be converted to a {@code LocalDate}.
259      * <p>
260      * This method matches the signature of the functional interface {@link TemporalQuery}
261      * allowing it to be used as a query via method reference, {@code MonthDay::from}.
262      *
263      * @param temporal the temporal object to convert, not null
264      * @return the month-day, not null
265      * @throws DateTimeException if unable to convert to a {@code MonthDay}
266      */
267     public static MonthDay from(TemporalAccessor temporal) {
268         if (temporal instanceof MonthDay) {

```

```

269         return (MonthDay) temporal;
270     }
271     try {
272         if (IsoChronology.INSTANCE.equals(Chronology.from(temporal)) == false) {
273             temporal = LocalDate.from(temporal);
274         }
275         return of(temporal.get(MONTH_OF_YEAR), temporal.get(DAY_OF_MONTH));
276     } catch (DateTimeException ex) {
277         throw new DateTimeException("Unable to obtain MonthDay from TemporalAccessor: " +
278             temporal + " of type " + temporal.getClass().getName(), ex);
279     }
280 }
281
282 //-----
283 /**
284  * Obtains an instance of {@code MonthDay} from a text string such as {@code --12-03}.
285  * <p>
286  * The string must represent a valid month-day.
287  * The format is {@code --MM-dd}.
288  *
289  * @param text the text to parse such as "--12-03", not null
290  * @return the parsed month-day, not null
291  * @throws DateTimeParseException if the text cannot be parsed
292  */
293 public static MonthDay parse(CharSequence text) {
294     return parse(text, PARSER);
295 }
296
297 /**
298  * Obtains an instance of {@code MonthDay} from a text string using a specific formatter.
299  * <p>
300  * The text is parsed using the formatter, returning a month-day.
301  *
302  * @param text the text to parse, not null
303  * @param formatter the formatter to use, not null
304  * @return the parsed month-day, not null
305  * @throws DateTimeParseException if the text cannot be parsed
306  */
307 public static MonthDay parse(CharSequence text, DateTimeFormatter formatter) {
308     Objects.requireNonNull(formatter, "formatter");
309     return formatter.parse(text, MonthDay::from);
310 }
311
312 //-----
313 /**
314  * Constructor, previously validated.
315  *
316  * @param month the month-of-year to represent, validated from 1 to 12
317  * @param dayOfMonth the day-of-month to represent, validated from 1 to 29-31
318  */
319 private MonthDay(int month, int dayOfMonth) {
320     this.month = month;
321     this.day = dayOfMonth;

```



```

322     }
323
324     //-----
325     /**
326      * Checks if the specified field is supported.
327      * <p>
328      * This checks if this month-day can be queried for the specified field.
329      * If false, then calling the {@link #range(TemporalField) range} and
330      * {@link #get(TemporalField) get} methods will throw an exception.
331      * <p>
332      * If the field is a {@link ChronoField} then the query is implemented here.
333      * The supported fields are:
334      * <ul>
335      * <li>{@code MONTH_OF_YEAR}
336      * <li>{@code YEAR}
337      * </ul>
338      * All other {@code ChronoField} instances will return false.
339      * <p>
340      * If the field is not a {@code ChronoField}, then the result of this method
341      * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
342      * passing {@code this} as the argument.
343      * Whether the field is supported is determined by the field.
344      *
345      * @param field the field to check, null returns false
346      * @return true if the field is supported on this month-day, false if not
347      */
348     @Override
349     public boolean isSupported(TemporalField field) {
350         if (field instanceof ChronoField) {
351             return field == MONTH_OF_YEAR || field == DAY_OF_MONTH;
352         }
353         return field != null && field.isSupportedBy(this);
354     }
355
356     /**
357      * Gets the range of valid values for the specified field.
358      * <p>
359      * The range object expresses the minimum and maximum valid values for a field.
360      * This month-day is used to enhance the accuracy of the returned range.
361      * If it is not possible to return the range, because the field is not supported
362      * or for some other reason, an exception is thrown.
363      * <p>
364      * If the field is a {@link ChronoField} then the query is implemented here.
365      * The {@link #isSupported(TemporalField) supported fields} will return
366      * appropriate range instances.
367      * All other {@code ChronoField} instances will throw an {@code
368          UnsupportedTemporalTypeException}.
369      * <p>
370      * If the field is not a {@code ChronoField}, then the result of this method
371      * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
372      * passing {@code this} as the argument.
373      * Whether the range can be obtained is determined by the field.
374      *

```

```

374     * @param field the field to query the range for, not null
375     * @return the range of valid values for the field, not null
376     * @throws DateTimeException if the range for the field cannot be obtained
377     * @throws UnsupportedTemporalTypeException if the field is not supported
378     */
379     @Override
380     public ValueRange range(TemporalField field) {
381         if (field == MONTH_OF_YEAR) {
382             return field.range();
383         } else if (field == DAY_OF_MONTH) {
384             return ValueRange.of(1, getMonth().minLength(), getMonth().maxLength());
385         }
386         return TemporalAccessor.super.range(field);
387     }
388
389     /**
390     * Gets the value of the specified field from this month-day as an {@code int}.
391     * <p>
392     * This queries this month-day for the value of the specified field.
393     * The returned value will always be within the valid range of values for the field.
394     * If it is not possible to return the value, because the field is not supported
395     * or for some other reason, an exception is thrown.
396     * <p>
397     * If the field is a {@link ChronoField} then the query is implemented here.
398     * The {@link #isSupported(TemporalField) supported fields} will return valid
399     * values based on this month-day.
400     * All other {@code ChronoField} instances will throw an {@code
401         UnsupportedTemporalTypeException}.
402     * <p>
403     * If the field is not a {@code ChronoField}, then the result of this method
404     * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
405     * passing {@code this} as the argument. Whether the value can be obtained,
406     * and what the value represents, is determined by the field.
407     *
408     * @param field the field to get, not null
409     * @return the value for the field
410     * @throws DateTimeException if a value for the field cannot be obtained or
411     *         the value is outside the range of valid values for the field
412     * @throws UnsupportedTemporalTypeException if the field is not supported or
413     *         the range of values exceeds an {@code int}
414     * @throws ArithmeticException if numeric overflow occurs
415     */
416     @Override // override for Javadoc
417     public int get(TemporalField field) {
418         return range(field).checkValidIntValue(getLong(field), field);
419     }
420
421     /**
422     * Gets the value of the specified field from this month-day as a {@code long}.
423     * <p>
424     * This queries this month-day for the value of the specified field.
425     * If it is not possible to return the value, because the field is not supported
426     * or for some other reason, an exception is thrown.

```

```

426 * <p>
427 * If the field is a {@link ChronoField} then the query is implemented here.
428 * The {@link #isSupported(TemporalField)} supported fields} will return valid
429 * values based on this month-day.
430 * All other {@code ChronoField} instances will throw an {@code
    UnsupportedTemporalTypeException}.
431 * <p>
432 * If the field is not a {@code ChronoField}, then the result of this method
433 * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
434 * passing {@code this} as the argument. Whether the value can be obtained,
435 * and what the value represents, is determined by the field.
436 *
437 * @param field the field to get, not null
438 * @return the value for the field
439 * @throws DateTimeException if a value for the field cannot be obtained
440 * @throws UnsupportedTemporalTypeException if the field is not supported
441 * @throws ArithmeticException if numeric overflow occurs
442 */
443 @Override
444 public long getLong(TemporalField field) {
445     if (field instanceof ChronoField) {
446         switch ((ChronoField) field) {
447             // alignedDOW and alignedWOM not supported because they cannot be set in with()
448             case DAY_OF_MONTH: return day;
449             case MONTH_OF_YEAR: return month;
450         }
451         throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
452     }
453     return field.getFrom(this);
454 }
455
456 //-----
457 /**
458 * Gets the month-of-year field from 1 to 12.
459 * <p>
460 * This method returns the month as an {@code int} from 1 to 12.
461 * Application code is frequently clearer if the enum {@link Month}
462 * is used by calling {@link #getMonth()}.
463 *
464 * @return the month-of-year, from 1 to 12
465 * @see #getMonth()
466 */
467 public int getMonthValue() {
468     return month;
469 }
470
471 /**
472 * Gets the month-of-year field using the {@code Month} enum.
473 * <p>
474 * This method returns the enum {@link Month} for the month.
475 * This avoids confusion as to what {@code int} values mean.
476 * If you need access to the primitive {@code int} value then the enum
477 * provides the {@link Month#getValue() int} value}.

```

```

478     *
479     * @return the month-of-year, not null
480     * @see #getMonthValue()
481     */
482     public Month getMonth() {
483         return Month.of(month);
484     }
485
486     /**
487     * Gets the day-of-month field.
488     * <p>
489     * This method returns the primitive {@code int} value for the day-of-month.
490     *
491     * @return the day-of-month, from 1 to 31
492     */
493     public int getDayOfMonth() {
494         return day;
495     }
496
497     //-----
498     /**
499     * Checks if the year is valid for this month-day.
500     * <p>
501     * This method checks whether this month and day and the input year form
502     * a valid date. This can only return false for February 29th.
503     *
504     * @param year the year to validate
505     * @return true if the year is valid for this month-day
506     * @see Year#isValidMonthDay(MonthDay)
507     */
508     public boolean isValidYear(int year) {
509         return (day == 29 && month == 2 && Year.isLeap(year) == false) == false;
510     }
511
512     //-----
513     /**
514     * Returns a copy of this {@code MonthDay} with the month-of-year altered.
515     * <p>
516     * This returns a month-day with the specified month.
517     * If the day-of-month is invalid for the specified month, the day will
518     * be adjusted to the last valid day-of-month.
519     * <p>
520     * This instance is immutable and unaffected by this method call.
521     *
522     * @param month the month-of-year to set in the returned month-day, from 1 (January) to 12 (
523         December)
524     * @return a {@code MonthDay} based on this month-day with the requested month, not null
525     * @throws DateTimeException if the month-of-year value is invalid
526     */
527     public MonthDay withMonth(int month) {
528         return with(Month.of(month));
529     }

```

```

530 /**
531  * Returns a copy of this {@code MonthDay} with the month-of-year altered.
532  * <p>
533  * This returns a month-day with the specified month.
534  * If the day-of-month is invalid for the specified month, the day will
535  * be adjusted to the last valid day-of-month.
536  * <p>
537  * This instance is immutable and unaffected by this method call.
538  *
539  * @param month the month-of-year to set in the returned month-day, not null
540  * @return a {@code MonthDay} based on this month-day with the requested month, not null
541  */
542 public MonthDay with(Month month) {
543     Objects.requireNonNull(month, "month");
544     if (month.getValue() == this.month) {
545         return this;
546     }
547     int day = Math.min(this.day, month.maxLength());
548     return new MonthDay(month.getValue(), day);
549 }
550
551 /**
552  * Returns a copy of this {@code MonthDay} with the day-of-month altered.
553  * <p>
554  * This returns a month-day with the specified day-of-month.
555  * If the day-of-month is invalid for the month, an exception is thrown.
556  * <p>
557  * This instance is immutable and unaffected by this method call.
558  *
559  * @param dayOfMonth the day-of-month to set in the return month-day, from 1 to 31
560  * @return a {@code MonthDay} based on this month-day with the requested day, not null
561  * @throws DateTimeException if the day-of-month value is invalid,
562  * or if the day-of-month is invalid for the month
563  */
564 public MonthDay withDayOfMonth(int dayOfMonth) {
565     if (dayOfMonth == this.day) {
566         return this;
567     }
568     return of(month, dayOfMonth);
569 }
570
571 //-----
572 /**
573  * Queries this month-day using the specified query.
574  * <p>
575  * This queries this month-day using the specified query strategy object.
576  * The {@code TemporalQuery} object defines the logic to be used to
577  * obtain the result. Read the documentation of the query to understand
578  * what the result of this method will be.
579  * <p>
580  * The result of this method is obtained by invoking the
581  * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
582  * specified query passing {@code this} as the argument.

```

```

583     *
584     * @param <R> the type of the result
585     * @param query the query to invoke, not null
586     * @return the query result, null may be returned (defined by the query)
587     * @throws DateTimeException if unable to query (defined by the query)
588     * @throws ArithmeticException if numeric overflow occurs (defined by the query)
589     */
590     @SuppressWarnings("unchecked")
591     @Override
592     public <R> R query(TemporalQuery<R> query) {
593         if (query == TemporalQueries.chronology()) {
594             return (R) IsoChronology.INSTANCE;
595         }
596         return TemporalAccessor.super.query(query);
597     }
598
599     /**
600     * Adjusts the specified temporal object to have this month-day.
601     * <p>
602     * This returns a temporal object of the same observable type as the input
603     * with the month and day-of-month changed to be the same as this.
604     * <p>
605     * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
606     * twice, passing {@link ChronoField#MONTH_OF_YEAR} and
607     * {@link ChronoField#DAY_OF_MONTH} as the fields.
608     * If the specified temporal object does not use the ISO calendar system then
609     * a {@code DateTimeException} is thrown.
610     * <p>
611     * In most cases, it is clearer to reverse the calling pattern by using
612     * {@link Temporal#with(TemporalAdjuster)}:
613     * <pre>
614     * // these two lines are equivalent, but the second approach is recommended
615     * temporal = thisMonthDay.adjustInto(temporal);
616     * temporal = temporal.with(thisMonthDay);
617     * </pre>
618     * <p>
619     * This instance is immutable and unaffected by this method call.
620     *
621     * @param temporal the target object to be adjusted, not null
622     * @return the adjusted object, not null
623     * @throws DateTimeException if unable to make the adjustment
624     * @throws ArithmeticException if numeric overflow occurs
625     */
626     @Override
627     public Temporal adjustInto(Temporal temporal) {
628         if (Chronology.from(temporal).equals(IsoChronology.INSTANCE) == false) {
629             throw new DateTimeException("Adjustment only supported on ISO date-time");
630         }
631         temporal = temporal.with(MONTH_OF_YEAR, month);
632         return temporal.with(DAY_OF_MONTH, Math.min(temporal.range(DAY_OF_MONTH).getMaximum(), day)
633             );
634     }

```

```

635  /**
636   * Formats this month-day using the specified formatter.
637   * <p>
638   * This month-day will be passed to the formatter to produce a string.
639   *
640   * @param formatter the formatter to use, not null
641   * @return the formatted month-day string, not null
642   * @throws DateTimeException if an error occurs during printing
643   */
644  public String format(DateTimeFormatter formatter) {
645      Objects.requireNonNull(formatter, "formatter");
646      return formatter.format(this);
647  }
648
649  //-----
650  /**
651   * Combines this month-day with a year to create a {@code LocalDate}.
652   * <p>
653   * This returns a {@code LocalDate} formed from this month-day and the specified year.
654   * <p>
655   * A month-day of February 29th will be adjusted to February 28th in the resulting
656   * date if the year is not a leap year.
657   * <p>
658   * This instance is immutable and unaffected by this method call.
659   *
660   * @param year the year to use, from MIN_YEAR to MAX_YEAR
661   * @return the local date formed from this month-day and the specified year, not null
662   * @throws DateTimeException if the year is outside the valid range of years
663   */
664  public LocalDate atYear(int year) {
665      return LocalDate.of(year, month, isValidYear(year) ? day : 28);
666  }
667
668  //-----
669  /**
670   * Compares this month-day to another month-day.
671   * <p>
672   * The comparison is based first on value of the month, then on the value of the day.
673   * It is "consistent with equals", as defined by {@link Comparable}.
674   *
675   * @param other the other month-day to compare to, not null
676   * @return the comparator value, negative if less, positive if greater
677   */
678  @Override
679  public int compareTo(MonthDay other) {
680      int cmp = (month - other.month);
681      if (cmp == 0) {
682          cmp = (day - other.day);
683      }
684      return cmp;
685  }
686
687  /**

```

```

688     * Checks if this month-day is after the specified month-day.
689     *
690     * @param other the other month-day to compare to, not null
691     * @return true if this is after the specified month-day
692     */
693     public boolean isAfter(MonthDay other) {
694         return compareTo(other) > 0;
695     }
696
697     /**
698     * Checks if this month-day is before the specified month-day.
699     *
700     * @param other the other month-day to compare to, not null
701     * @return true if this point is before the specified month-day
702     */
703     public boolean isBefore(MonthDay other) {
704         return compareTo(other) < 0;
705     }
706
707     //-----
708     /**
709     * Checks if this month-day is equal to another month-day.
710     * <p>
711     * The comparison is based on the time-line position of the month-day within a year.
712     *
713     * @param obj the object to check, null returns false
714     * @return true if this is equal to the other month-day
715     */
716     @Override
717     public boolean equals(Object obj) {
718         if (this == obj) {
719             return true;
720         }
721         if (obj instanceof MonthDay) {
722             MonthDay other = (MonthDay) obj;
723             return month == other.month && day == other.day;
724         }
725         return false;
726     }
727
728     /**
729     * A hash code for this month-day.
730     *
731     * @return a suitable hash code
732     */
733     @Override
734     public int hashCode() {
735         return (month << 6) + day;
736     }
737
738     //-----
739     /**
740     * Outputs this month-day as a {@code String}, such as {@code --12-03}.

```



```

741     * <p>
742     * The output will be in the format {@code --MM-dd}:
743     *
744     * @return a string representation of this month-day, not null
745     */
746     @Override
747     public String toString() {
748         return new StringBuilder(10).append("--")
749             .append(month < 10 ? "0" : "").append(month)
750             .append(day < 10 ? "-0" : "-").append(day)
751             .toString();
752     }
753
754     //-----
755     /**
756     * Writes the object using a
757     * <a href="../../serialized-form.html#java.time.Ser">dedicated serialized form</a>.
758     * @serialData
759     * <pre>
760     *   out.writeByte(13); // identifies a MonthDay
761     *   out.writeByte(month);
762     *   out.writeByte(day);
763     * </pre>
764     *
765     * @return the instance of {@code Ser}, not null
766     */
767     private Object writeReplace() {
768         return new Ser(Ser.MONTH_DAY_TYPE, this);
769     }
770
771     /**
772     * Defend against malicious streams.
773     *
774     * @param s the stream to read
775     * @throws InvalidObjectException always
776     */
777     private void readObject(ObjectInputStream s) throws InvalidObjectException {
778         throw new InvalidObjectException("Deserialization via serialization delegate");
779     }
780
781     void writeExternal(DataOutput out) throws IOException {
782         out.writeByte(month);
783         out.writeByte(day);
784     }
785
786     static MonthDay readExternal(DataInput in) throws IOException {
787         byte month = in.readByte();
788         byte day = in.readByte();
789         return MonthDay.of(month, day);
790     }
791
792 }

```

4.11 OffsetDateTime.java

Secuencia 60: java/time/OffsetDateTime.java

```
1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
```

```

51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time;
63
64  import static java.time.temporal.ChronoField.EPOCH_DAY;
65  import static java.time.temporal.ChronoField.INSTANT_SECONDS;
66  import static java.time.temporal.ChronoField.NANO_OF_DAY;
67  import static java.time.temporal.ChronoField.OFFSET_SECONDS;
68  import static java.time.temporal.ChronoUnit.FOREVER;
69  import static java.time.temporal.ChronoUnit.NANOS;
70
71  import java.io.IOException;
72  import java.io.ObjectInput;
73  import java.io.ObjectOutput;
74  import java.io.InvalidObjectException;
75  import java.io.ObjectInputStream;
76  import java.io.Serializable;
77  import java.time.chrono.IsoChronology;
78  import java.time.format.DateTimeFormatter;
79  import java.time.format.DateTimeParseException;
80  import java.time.temporal.ChronoField;
81  import java.time.temporal.ChronoUnit;
82  import java.time.temporal.Temporal;
83  import java.time.temporal.TemporalAccessor;
84  import java.time.temporal.TemporalAdjuster;
85  import java.time.temporal.TemporalAmount;
86  import java.time.temporal.TemporalField;
87  import java.time.temporal.TemporalQueries;
88  import java.time.temporal.TemporalQuery;
89  import java.time.temporal.TemporalUnit;
90  import java.time.temporal.UnsupportedTemporalTypeException;
91  import java.time.temporal.ValueRange;
92  import java.time.zone.ZoneRules;
93  import java.util.Comparator;
94  import java.util.Objects;
95
96  /**
97   * A date-time with an offset from UTC/Greenwich in the ISO-8601 calendar system,
98   * such as {@code 2007-12-03T10:15:30+01:00}.
99   * <p>
100  * {@code OffsetDateTime} is an immutable representation of a date-time with an offset.
101  * This class stores all date and time fields, to a precision of nanoseconds,
102  * as well as the offset from UTC/Greenwich. For example, the value
103  * "2nd October 2007 at 13:45.30.123456789 +02:00" can be stored in an {@code OffsetDateTime}.

```

```

104 * <p>
105 * {@code OffsetDateTime}, {@link java.time.ZonedDateTime} and {@link java.time.Instant} all store
    an instant
106 * on the time-line to nanosecond precision.
107 * {@code Instant} is the simplest, simply representing the instant.
108 * {@code OffsetDateTime} adds to the instant the offset from UTC/Greenwich, which allows
109 * the local date-time to be obtained.
110 * {@code ZonedDateTime} adds full time-zone rules.
111 * <p>
112 * It is intended that {@code ZonedDateTime} or {@code Instant} is used to model data
113 * in simpler applications. This class may be used when modeling date-time concepts in
114 * more detail, or when communicating to a database or in a network protocol.
115 *
116 * <p>
117 * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
118 * class; use of identity-sensitive operations (including reference equality
119 * ({@code ==}), identity hash code, or synchronization) on instances of
120 * {@code OffsetDateTime} may have unpredictable results and should be avoided.
121 * The {@code equals} method should be used for comparisons.
122 *
123 * @implSpec
124 * This class is immutable and thread-safe.
125 *
126 * @since 1.8
127 */
128 public final class OffsetDateTime
129     implements Temporal, TemporalAdjuster, Comparable<OffsetDateTime>, Serializable {
130
131     /**
132      * The minimum supported {@code OffsetDateTime}, '-999999999-01-01T00:00:00+18:00'.
133      * This is the local date-time of midnight at the start of the minimum date
134      * in the maximum offset (larger offsets are earlier on the time-line).
135      * This combines {@link LocalDateTime#MIN} and {@link ZoneOffset#MAX}.
136      * This could be used by an application as a "far past" date-time.
137      */
138     public static final OffsetDateTime MIN = LocalDateTime.MIN.atOffset(ZoneOffset.MAX);
139
140     /**
141      * The maximum supported {@code OffsetDateTime}, '+999999999-12-31T23:59:59.999999999-18:00'.
142      * This is the local date-time just before midnight at the end of the maximum date
143      * in the minimum offset (larger negative offsets are later on the time-line).
144      * This combines {@link LocalDateTime#MAX} and {@link ZoneOffset#MIN}.
145      * This could be used by an application as a "far future" date-time.
146      */
147     public static final OffsetDateTime MAX = LocalDateTime.MAX.atOffset(ZoneOffset.MIN);
148
149     /**
150      * Gets a comparator that compares two {@code OffsetDateTime} instances
151      * based solely on the instant.
152      * <p>
153      * This method differs from the comparison in {@link #compareTo} in that it
154      * only compares the underlying instant.
155      *
156      * @return a comparator that compares in time-line order

```

```

156     *
157     * @see #isAfter
158     * @see #isBefore
159     * @see #isEqual
160     */
161     public static Comparator<OffsetDateTime> timeLineOrder() {
162         return OffsetDateTime::compareInstant;
163     }
164
165     /**
166     * Compares this {@code OffsetDateTime} to another date-time.
167     * The comparison is based on the instant.
168     *
169     * @param datetime1 the first date-time to compare, not null
170     * @param datetime2 the other date-time to compare to, not null
171     * @return the comparator value, negative if less, positive if greater
172     */
173     private static int compareInstant(OffsetDateTime datetime1, OffsetDateTime datetime2) {
174         if (datetime1.getOffset().equals(datetime2.getOffset())) {
175             return datetime1.toLocalDateTime().compareTo(datetime2.toLocalDateTime());
176         }
177         int cmp = Long.compare(datetime1.toEpochSecond(), datetime2.toEpochSecond());
178         if (cmp == 0) {
179             cmp = datetime1.toLocalTime().getNano() - datetime2.toLocalTime().getNano();
180         }
181         return cmp;
182     }
183
184     /**
185     * Serialization version.
186     */
187     private static final long serialVersionUID = 2287754244819255394L;
188
189     /**
190     * The local date-time.
191     */
192     private final LocalDateTime dateTime;
193
194     /**
195     * The offset from UTC/Greenwich.
196     */
197     private final ZoneOffset offset;
198
199     //-----
200     /**
201     * Obtains the current date-time from the system clock in the default time-zone.
202     * <p>
203     * This will query the {@link Clock#systemDefaultZone() system clock} in the default
204     * time-zone to obtain the current date-time.
205     * The offset will be calculated from the time-zone in the clock.
206     * <p>
207     * Using this method will prevent the ability to use an alternate clock for testing
208     * because the clock is hard-coded.
209     *

```

```

209     * @return the current date-time using the system clock, not null
210     */
211     public static OffsetDateTime now() {
212         return now(Clock.systemDefaultZone());
213     }
214
215     /**
216     * Obtains the current date-time from the system clock in the specified time-zone.
217     * <p>
218     * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current date-
219     * time.
220     * Specifying the time-zone avoids dependence on the default time-zone.
221     * The offset will be calculated from the specified time-zone.
222     * <p>
223     * Using this method will prevent the ability to use an alternate clock for testing
224     * because the clock is hard-coded.
225     *
226     * @param zone the zone ID to use, not null
227     * @return the current date-time using the system clock, not null
228     */
229     public static OffsetDateTime now(ZoneId zone) {
230         return now(Clock.system(zone));
231     }
232
233     /**
234     * Obtains the current date-time from the specified clock.
235     * <p>
236     * This will query the specified clock to obtain the current date-time.
237     * The offset will be calculated from the time-zone in the clock.
238     * <p>
239     * Using this method allows the use of an alternate clock for testing.
240     * The alternate clock may be introduced using {@link Clock dependency injection}.
241     *
242     * @param clock the clock to use, not null
243     * @return the current date-time, not null
244     */
245     public static OffsetDateTime now(Clock clock) {
246         Objects.requireNonNull(clock, "clock");
247         final Instant now = clock.instant(); // called once
248         return ofInstant(now, clock.getZone().getRules().getOffset(now));
249     }
250
251     //-----
252     /**
253     * Obtains an instance of {@code OffsetDateTime} from a date, time and offset.
254     * <p>
255     * This creates an offset date-time with the specified local date, time and offset.
256     *
257     * @param date the local date, not null
258     * @param time the local time, not null
259     * @param offset the zone offset, not null
260     * @return the offset date-time, not null
261     */

```

```

261 public static OffsetDateTime of(LocalDate date, LocalTime time, ZoneOffset offset) {
262     LocalDateTime dt = LocalDateTime.of(date, time);
263     return new OffsetDateTime(dt, offset);
264 }
265
266 /**
267  * Obtains an instance of {@code OffsetDateTime} from a date-time and offset.
268  * <p>
269  * This creates an offset date-time with the specified local date-time and offset.
270  *
271  * @param dateTime the local date-time, not null
272  * @param offset the zone offset, not null
273  * @return the offset date-time, not null
274  */
275 public static OffsetDateTime of(LocalDateTime dateTime, ZoneOffset offset) {
276     return new OffsetDateTime(dateTime, offset);
277 }
278
279 /**
280  * Obtains an instance of {@code OffsetDateTime} from a year, month, day,
281  * hour, minute, second, nanosecond and offset.
282  * <p>
283  * This creates an offset date-time with the seven specified fields.
284  * <p>
285  * This method exists primarily for writing test cases.
286  * Non test-code will typically use other methods to create an offset time.
287  * {@code LocalDateTime} has five additional convenience variants of the
288  * equivalent factory method taking fewer arguments.
289  * They are not provided here to reduce the footprint of the API.
290  *
291  * @param year the year to represent, from MIN_YEAR to MAX_YEAR
292  * @param month the month-of-year to represent, from 1 (January) to 12 (December)
293  * @param dayOfMonth the day-of-month to represent, from 1 to 31
294  * @param hour the hour-of-day to represent, from 0 to 23
295  * @param minute the minute-of-hour to represent, from 0 to 59
296  * @param second the second-of-minute to represent, from 0 to 59
297  * @param nanoOfSecond the nano-of-second to represent, from 0 to 999,999,999
298  * @param offset the zone offset, not null
299  * @return the offset date-time, not null
300  * @throws DateTimeException if the value of any field is out of range, or
301  * if the day-of-month is invalid for the month-year
302  */
303 public static OffsetDateTime of(
304     int year, int month, int dayOfMonth,
305     int hour, int minute, int second, int nanoOfSecond, ZoneOffset offset) {
306     LocalDateTime dt = LocalDateTime.of(year, month, dayOfMonth, hour, minute, second,
307         nanoOfSecond);
308     return new OffsetDateTime(dt, offset);
309 }
310
311 //-----
312 /**
313  * Obtains an instance of {@code OffsetDateTime} from an {@code Instant} and zone ID.

```

```

313 * <p>
314 * This creates an offset date-time with the same instant as that specified.
315 * Finding the offset from UTC/Greenwich is simple as there is only one valid
316 * offset for each instant.
317 *
318 * @param instant the instant to create the date-time from, not null
319 * @param zone the time-zone, which may be an offset, not null
320 * @return the offset date-time, not null
321 * @throws DateTimeException if the result exceeds the supported range
322 */
323 public static OffsetDateTime ofInstant(Instant instant, ZoneId zone) {
324     Objects.requireNonNull(instant, "instant");
325     Objects.requireNonNull(zone, "zone");
326     ZoneRules rules = zone.getRules();
327     ZoneOffset offset = rules.getOffset(instant);
328     LocalDateTime ldt = LocalDateTime.ofEpochSecond(instant.getEpochSecond(), instant.getNano()
329         , offset);
330     return new OffsetDateTime(ldt, offset);
331 }
332 //-----
333 /**
334 * Obtains an instance of {@code OffsetDateTime} from a temporal object.
335 * <p>
336 * This obtains an offset date-time based on the specified temporal.
337 * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
338 * which this factory converts to an instance of {@code OffsetDateTime}.
339 * <p>
340 * The conversion will first obtain a {@code ZoneOffset} from the temporal object.
341 * It will then try to obtain a {@code LocalDateTime}, falling back to an {@code Instant} if
342     necessary.
343 * The result will be the combination of {@code ZoneOffset} with either
344 * with {@code LocalDateTime} or {@code Instant}.
345 * Implementations are permitted to perform optimizations such as accessing
346 * those fields that are equivalent to the relevant objects.
347 * <p>
348 * This method matches the signature of the functional interface {@link TemporalQuery}
349 * allowing it to be used as a query via method reference, {@code OffsetDateTime::from}.
350 *
351 * @param temporal the temporal object to convert, not null
352 * @return the offset date-time, not null
353 * @throws DateTimeException if unable to convert to an {@code OffsetDateTime}
354 */
355 public static OffsetDateTime from(TemporalAccessor temporal) {
356     if (temporal instanceof OffsetDateTime) {
357         return (OffsetDateTime) temporal;
358     }
359     try {
360         ZoneOffset offset = ZoneOffset.from(temporal);
361         LocalDate date = temporal.query(TemporalQueries.localDate());
362         LocalTime time = temporal.query(TemporalQueries.localTime());
363         if (date != null && time != null) {
364             return OffsetDateTime.of(date, time, offset);
365         }
366     }

```



```

364         } else {
365             Instant instant = Instant.from(temporal);
366             return OffsetDateTime.ofInstant(instant, offset);
367         }
368     } catch (DateTimeException ex) {
369         throw new DateTimeException("Unable to obtain OffsetDateTime from TemporalAccessor: " +
370             temporal + " of type " + temporal.getClass().getName(), ex);
371     }
372 }
373
374 //-----
375 /**
376  * Obtains an instance of {@code OffsetDateTime} from a text string
377  * such as {@code 2007-12-03T10:15:30+01:00}.
378  * <p>
379  * The string must represent a valid date-time and is parsed using
380  * {@link java.time.format.DateTimeFormatter#ISO_OFFSET_DATE_TIME}.
381  *
382  * @param text the text to parse such as "2007-12-03T10:15:30+01:00", not null
383  * @return the parsed offset date-time, not null
384  * @throws DateTimeParseException if the text cannot be parsed
385  */
386 public static OffsetDateTime parse(CharSequence text) {
387     return parse(text, DateTimeFormatter.ISO_OFFSET_DATE_TIME);
388 }
389
390 /**
391  * Obtains an instance of {@code OffsetDateTime} from a text string using a specific formatter.
392  * <p>
393  * The text is parsed using the formatter, returning a date-time.
394  *
395  * @param text the text to parse, not null
396  * @param formatter the formatter to use, not null
397  * @return the parsed offset date-time, not null
398  * @throws DateTimeParseException if the text cannot be parsed
399  */
400 public static OffsetDateTime parse(CharSequence text, DateTimeFormatter formatter) {
401     Objects.requireNonNull(formatter, "formatter");
402     return formatter.parse(text, OffsetDateTime::from);
403 }
404
405 //-----
406 /**
407  * Constructor.
408  *
409  * @param dateTime the local date-time, not null
410  * @param offset the zone offset, not null
411  */
412 private OffsetDateTime(LocalDateTime dateTime, ZoneOffset offset) {
413     this.dateTime = Objects.requireNonNull(dateTime, "dateTime");
414     this.offset = Objects.requireNonNull(offset, "offset");
415 }
416

```

```

417 /**
418  * Returns a new date-time based on this one, returning {@code this} where possible.
419  *
420  * @param dateTime the date-time to create with, not null
421  * @param offset the zone offset to create with, not null
422  */
423 private OffsetDateTime with(LocalDateTime dateTime, ZoneOffset offset) {
424     if (this.dateTime == dateTime && this.offset.equals(offset)) {
425         return this;
426     }
427     return new OffsetDateTime(dateTime, offset);
428 }
429
430 //-----
431 /**
432  * Checks if the specified field is supported.
433  * <p>
434  * This checks if this date-time can be queried for the specified field.
435  * If false, then calling the {@link #range(TemporalField) range},
436  * {@link #get(TemporalField) get} and {@link #with(TemporalField, long)}
437  * methods will throw an exception.
438  * <p>
439  * If the field is a {@link ChronoField} then the query is implemented here.
440  * The supported fields are:
441  * <ul>
442  * <li>{@code NANO_OF_SECOND}
443  * <li>{@code NANO_OF_DAY}
444  * <li>{@code MICRO_OF_SECOND}
445  * <li>{@code MICRO_OF_DAY}
446  * <li>{@code MILLI_OF_SECOND}
447  * <li>{@code MILLI_OF_DAY}
448  * <li>{@code SECOND_OF_MINUTE}
449  * <li>{@code SECOND_OF_DAY}
450  * <li>{@code MINUTE_OF_HOUR}
451  * <li>{@code MINUTE_OF_DAY}
452  * <li>{@code HOUR_OF_AMPM}
453  * <li>{@code CLOCK_HOUR_OF_AMPM}
454  * <li>{@code HOUR_OF_DAY}
455  * <li>{@code CLOCK_HOUR_OF_DAY}
456  * <li>{@code AMPM_OF_DAY}
457  * <li>{@code DAY_OF_WEEK}
458  * <li>{@code ALIGNED_DAY_OF_WEEK_IN_MONTH}
459  * <li>{@code ALIGNED_DAY_OF_WEEK_IN_YEAR}
460  * <li>{@code DAY_OF_MONTH}
461  * <li>{@code DAY_OF_YEAR}
462  * <li>{@code EPOCH_DAY}
463  * <li>{@code ALIGNED_WEEK_OF_MONTH}
464  * <li>{@code ALIGNED_WEEK_OF_YEAR}
465  * <li>{@code MONTH_OF_YEAR}
466  * <li>{@code PROLEPTIC_MONTH}
467  * <li>{@code YEAR_OF_ERA}
468  * <li>{@code YEAR}
469  * <li>{@code ERA}

```

```

470 * <li>{@code INSTANT_SECONDS}
471 * <li>{@code OFFSET_SECONDS}
472 * </ul>
473 * All other {@code ChronoField} instances will return false.
474 * <p>
475 * If the field is not a {@code ChronoField}, then the result of this method
476 * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
477 * passing {@code this} as the argument.
478 * Whether the field is supported is determined by the field.
479 *
480 * @param field the field to check, null returns false
481 * @return true if the field is supported on this date-time, false if not
482 */
483 @Override
484 public boolean isSupported(TemporalField field) {
485     return field instanceof ChronoField || (field != null && field.isSupportedBy(this));
486 }
487
488 /**
489 * Checks if the specified unit is supported.
490 * <p>
491 * This checks if the specified unit can be added to, or subtracted from, this date-time.
492 * If false, then calling the {@link #plus(long, TemporalUnit)} and
493 * {@link #minus(long, TemporalUnit) minus} methods will throw an exception.
494 * <p>
495 * If the unit is a {@link ChronoUnit} then the query is implemented here.
496 * The supported units are:
497 * <ul>
498 * <li>{@code NANOS}
499 * <li>{@code MICROS}
500 * <li>{@code MILLIS}
501 * <li>{@code SECONDS}
502 * <li>{@code MINUTES}
503 * <li>{@code HOURS}
504 * <li>{@code HALF_DAYS}
505 * <li>{@code DAYS}
506 * <li>{@code WEEKS}
507 * <li>{@code MONTHS}
508 * <li>{@code YEARS}
509 * <li>{@code DECADES}
510 * <li>{@code CENTURIES}
511 * <li>{@code MILLENNIA}
512 * <li>{@code ERAS}
513 * </ul>
514 * All other {@code ChronoUnit} instances will return false.
515 * <p>
516 * If the unit is not a {@code ChronoUnit}, then the result of this method
517 * is obtained by invoking {@code TemporalUnit.isSupportedBy(Temporal)}
518 * passing {@code this} as the argument.
519 * Whether the unit is supported is determined by the unit.
520 *
521 * @param unit the unit to check, null returns false
522 * @return true if the unit can be added/subtracted, false if not

```

```

523     */
524     @Override // override for Javadoc
525     public boolean isSupported(TemporalUnit unit) {
526         if (unit instanceof ChronoUnit) {
527             return unit != FOREVER;
528         }
529         return unit != null && unit.isSupportedBy(this);
530     }
531
532     //-----
533     /**
534      * Gets the range of valid values for the specified field.
535      * <p>
536      * The range object expresses the minimum and maximum valid values for a field.
537      * This date-time is used to enhance the accuracy of the returned range.
538      * If it is not possible to return the range, because the field is not supported
539      * or for some other reason, an exception is thrown.
540      * <p>
541      * If the field is a {@link ChronoField} then the query is implemented here.
542      * The {@link #isSupported(TemporalField) supported fields} will return
543      * appropriate range instances.
544      * All other {@code ChronoField} instances will throw an {@code
545          UnsupportedTemporalTypeException}.
546      * <p>
547      * If the field is not a {@code ChronoField}, then the result of this method
548      * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
549      * passing {@code this} as the argument.
550      * Whether the range can be obtained is determined by the field.
551      *
552      * @param field the field to query the range for, not null
553      * @return the range of valid values for the field, not null
554      * @throws DateTimeException if the range for the field cannot be obtained
555      * @throws UnsupportedTemporalTypeException if the field is not supported
556      */
557     @Override
558     public ValueRange range(TemporalField field) {
559         if (field instanceof ChronoField) {
560             if (field == INSTANT_SECONDS || field == OFFSET_SECONDS) {
561                 return field.range();
562             }
563             return dateTime.range(field);
564         }
565         return field.rangeRefinedBy(this);
566     }
567
568     /**
569      * Gets the value of the specified field from this date-time as an {@code int}.
570      * <p>
571      * This queries this date-time for the value of the specified field.
572      * The returned value will always be within the valid range of values for the field.
573      * If it is not possible to return the value, because the field is not supported
574      * or for some other reason, an exception is thrown.
575      * <p>

```

```

575 * If the field is a {@link ChronoField} then the query is implemented here.
576 * The {@link #isSupported(TemporalField) supported fields} will return valid
577 * values based on this date-time, except {@code NANO_OF_DAY}, {@code MICRO_OF_DAY},
578 * {@code EPOCH_DAY}, {@code PROLEPTIC_MONTH} and {@code INSTANT_SECONDS} which are too
579 * large to fit in an {@code int} and throw a {@code DateTimeException}.
580 * All other {@code ChronoField} instances will throw an {@code
    UnsupportedTemporalTypeException}.
581 * <p>
582 * If the field is not a {@code ChronoField}, then the result of this method
583 * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
584 * passing {@code this} as the argument. Whether the value can be obtained,
585 * and what the value represents, is determined by the field.
586 *
587 * @param field the field to get, not null
588 * @return the value for the field
589 * @throws DateTimeException if a value for the field cannot be obtained or
590 *         the value is outside the range of valid values for the field
591 * @throws UnsupportedTemporalTypeException if the field is not supported or
592 *         the range of values exceeds an {@code int}
593 * @throws ArithmeticException if numeric overflow occurs
594 */
595 @Override
596 public int get(TemporalField field) {
597     if (field instanceof ChronoField) {
598         switch ((ChronoField) field) {
599             case INSTANT_SECONDS:
600                 throw new UnsupportedTemporalTypeException("Invalid field 'InstantSeconds' for
                    get() method, use getLong() instead");
601             case OFFSET_SECONDS:
602                 return getOffset().getTotalSeconds();
603         }
604         return dateTime.get(field);
605     }
606     return Temporal.super.get(field);
607 }
608
609 /**
610 * Gets the value of the specified field from this date-time as a {@code long}.
611 * <p>
612 * This queries this date-time for the value of the specified field.
613 * If it is not possible to return the value, because the field is not supported
614 * or for some other reason, an exception is thrown.
615 * <p>
616 * If the field is a {@link ChronoField} then the query is implemented here.
617 * The {@link #isSupported(TemporalField) supported fields} will return valid
618 * values based on this date-time.
619 * All other {@code ChronoField} instances will throw an {@code
    UnsupportedTemporalTypeException}.
620 * <p>
621 * If the field is not a {@code ChronoField}, then the result of this method
622 * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
623 * passing {@code this} as the argument. Whether the value can be obtained,
624 * and what the value represents, is determined by the field.

```

```

625     *
626     * @param field the field to get, not null
627     * @return the value for the field
628     * @throws DateTimeException if a value for the field cannot be obtained
629     * @throws UnsupportedTemporalTypeException if the field is not supported
630     * @throws ArithmeticException if numeric overflow occurs
631     */
632     @Override
633     public long getLong(TemporalField field) {
634         if (field instanceof ChronoField) {
635             switch ((ChronoField) field) {
636                 case INSTANT_SECONDS: return toEpochSecond();
637                 case OFFSET_SECONDS: return getOffset().getTotalSeconds();
638             }
639             return dateTime.getLong(field);
640         }
641         return field.getFrom(this);
642     }
643
644     //-----
645     /**
646     * Gets the zone offset, such as '+01:00'.
647     * <p>
648     * This is the offset of the local date-time from UTC/Greenwich.
649     *
650     * @return the zone offset, not null
651     */
652     public ZoneOffset getOffset() {
653         return offset;
654     }
655
656     /**
657     * Returns a copy of this {@code OffsetDateTime} with the specified offset ensuring
658     * that the result has the same local date-time.
659     * <p>
660     * This method returns an object with the same {@code LocalDateTime} and the specified {@code
661     * ZoneOffset}.
662     * No calculation is needed or performed.
663     * For example, if this time represents {@code 2007-12-03T10:30+02:00} and the offset specified
664     * is
665     * {@code +03:00}, then this method will return {@code 2007-12-03T10:30+03:00}.
666     * <p>
667     * To take into account the difference between the offsets, and adjust the time fields,
668     * use {@link #withOffsetSameInstant}.
669     * <p>
670     * This instance is immutable and unaffected by this method call.
671     *
672     * @param offset the zone offset to change to, not null
673     * @return an {@code OffsetDateTime} based on this date-time with the requested offset, not
674     *         null
675     */
676     public OffsetDateTime withOffsetSameLocal(ZoneOffset offset) {
677         return with(dateTime, offset);
678     }

```

```

675     }
676
677     /**
678      * Returns a copy of this {@code OffsetDateTime} with the specified offset ensuring
679      * that the result is at the same instant.
680      * <p>
681      * This method returns an object with the specified {@code ZoneOffset} and a {@code
682      * LocalDateTime}
683      * adjusted by the difference between the two offsets.
684      * This will result in the old and new objects representing the same instant.
685      * This is useful for finding the local time in a different offset.
686      * For example, if this time represents {@code 2007-12-03T10:30+02:00} and the offset specified
687      * is
688      * {@code +03:00}, then this method will return {@code 2007-12-03T11:30+03:00}.
689      * <p>
690      * To change the offset without adjusting the local time use {@link #withOffsetSameLocal}.
691      * <p>
692      * This instance is immutable and unaffected by this method call.
693      *
694      * @param offset the zone offset to change to, not null
695      * @return an {@code OffsetDateTime} based on this date-time with the requested offset, not
696      *         null
697      * @throws DateTimeException if the result exceeds the supported date range
698      */
699     public OffsetDateTime withOffsetSameInstant(ZoneOffset offset) {
700         if (offset.equals(this.offset)) {
701             return this;
702         }
703         int difference = offset.getTotalSeconds() - this.offset.getTotalSeconds();
704         LocalDateTime adjusted = dateTime.plusSeconds(difference);
705         return new OffsetDateTime(adjusted, offset);
706     }
707
708     //-----
709     /**
710      * Gets the {@code LocalDateTime} part of this date-time.
711      * <p>
712      * This returns a {@code LocalDateTime} with the same year, month, day and time
713      * as this date-time.
714      *
715      * @return the local date-time part of this date-time, not null
716      */
717     public LocalDateTime toLocalDateTime() {
718         return dateTime;
719     }
720
721     //-----
722     /**
723      * Gets the {@code LocalDate} part of this date-time.
724      * <p>
725      * This returns a {@code LocalDate} with the same year, month and day
726      * as this date-time.
727      *

```

```
725     * @return the date part of this date-time, not null
726     */
727     public LocalDate toLocalDate() {
728         return dateTime.toLocalDate();
729     }
730
731     /**
732     * Gets the year field.
733     * <p>
734     * This method returns the primitive {@code int} value for the year.
735     * <p>
736     * The year returned by this method is proleptic as per {@code get(YEAR)}.
737     * To obtain the year-of-era, use {@code get(YEAR_OF_ERA)}.
738     *
739     * @return the year, from MIN_YEAR to MAX_YEAR
740     */
741     public int getYear() {
742         return dateTime.getYear();
743     }
744
745     /**
746     * Gets the month-of-year field from 1 to 12.
747     * <p>
748     * This method returns the month as an {@code int} from 1 to 12.
749     * Application code is frequently clearer if the enum {@link Month}
750     * is used by calling {@link #getMonth()}.
751     *
752     * @return the month-of-year, from 1 to 12
753     * @see #getMonth()
754     */
755     public int getMonthValue() {
756         return dateTime.getMonthValue();
757     }
758
759     /**
760     * Gets the month-of-year field using the {@code Month} enum.
761     * <p>
762     * This method returns the enum {@link Month} for the month.
763     * This avoids confusion as to what {@code int} values mean.
764     * If you need access to the primitive {@code int} value then the enum
765     * provides the {@link Month#getValue() int value}.
766     *
767     * @return the month-of-year, not null
768     * @see #getMonthValue()
769     */
770     public Month getMonth() {
771         return dateTime.getMonth();
772     }
773
774     /**
775     * Gets the day-of-month field.
776     * <p>
777     * This method returns the primitive {@code int} value for the day-of-month.
```



```
778     *
779     * @return the day-of-month, from 1 to 31
780     */
781     public int getDayOfMonth() {
782         return dateTime.getDayOfMonth();
783     }
784
785     /**
786     * Gets the day-of-year field.
787     * <p>
788     * This method returns the primitive {@code int} value for the day-of-year.
789     *
790     * @return the day-of-year, from 1 to 365, or 366 in a leap year
791     */
792     public int getDayOfYear() {
793         return dateTime.getDayOfYear();
794     }
795
796     /**
797     * Gets the day-of-week field, which is an enum {@code DayOfWeek}.
798     * <p>
799     * This method returns the enum {@link DayOfWeek} for the day-of-week.
800     * This avoids confusion as to what {@code int} values mean.
801     * If you need access to the primitive {@code int} value then the enum
802     * provides the {@link DayOfWeek#getValue() int value}.
803     * <p>
804     * Additional information can be obtained from the {@code DayOfWeek}.
805     * This includes textual names of the values.
806     *
807     * @return the day-of-week, not null
808     */
809     public DayOfWeek getDayOfWeek() {
810         return dateTime.getDayOfWeek();
811     }
812
813     //-----
814     /**
815     * Gets the {@code LocalTime} part of this date-time.
816     * <p>
817     * This returns a {@code LocalTime} with the same hour, minute, second and
818     * nanosecond as this date-time.
819     *
820     * @return the time part of this date-time, not null
821     */
822     public LocalTime toLocalTime() {
823         return dateTime.toLocalTime();
824     }
825
826     /**
827     * Gets the hour-of-day field.
828     *
829     * @return the hour-of-day, from 0 to 23
830     */
```

```

831 public int getHour() {
832     return dateTime.getHour();
833 }
834
835 /**
836  * Gets the minute-of-hour field.
837  *
838  * @return the minute-of-hour, from 0 to 59
839  */
840 public int getMinute() {
841     return dateTime.getMinute();
842 }
843
844 /**
845  * Gets the second-of-minute field.
846  *
847  * @return the second-of-minute, from 0 to 59
848  */
849 public int getSecond() {
850     return dateTime.getSecond();
851 }
852
853 /**
854  * Gets the nano-of-second field.
855  *
856  * @return the nano-of-second, from 0 to 999,999,999
857  */
858 public int getNano() {
859     return dateTime.getNano();
860 }
861
862 //-----
863 /**
864  * Returns an adjusted copy of this date-time.
865  * <p>
866  * This returns an {@code OffsetDateTime}, based on this one, with the date-time adjusted.
867  * The adjustment takes place using the specified adjuster strategy object.
868  * Read the documentation of the adjuster to understand what adjustment will be made.
869  * <p>
870  * A simple adjuster might simply set the one of the fields, such as the year field.
871  * A more complex adjuster might set the date to the last day of the month.
872  * A selection of common adjustments is provided in
873  * {@link java.time.temporal.TemporalAdjusters TemporalAdjusters}.
874  * These include finding the "last day of the month" and "next Wednesday".
875  * Key date-time classes also implement the {@code TemporalAdjuster} interface,
876  * such as {@link Month} and {@link java.time.MonthDay MonthDay}.
877  * The adjuster is responsible for handling special cases, such as the varying
878  * lengths of month and leap years.
879  * <p>
880  * For example this code returns a date on the last day of July:
881  * <pre>
882  * import static java.time.Month.*;
883  * import static java.time.temporal.TemporalAdjusters.*;

```

```

884 *
885 * result = offsetDateTime.with(JULY).with(lastDayOfMonth());
886 * </pre>
887 * <p>
888 * The classes {@link LocalDate}, {@link LocalTime} and {@link ZoneOffset} implement
889 * {@code TemporalAdjuster}, thus this method can be used to change the date, time or offset:
890 * <pre>
891 * result = offsetDateTime.with(date);
892 * result = offsetDateTime.with(time);
893 * result = offsetDateTime.with(offset);
894 * </pre>
895 * <p>
896 * The result of this method is obtained by invoking the
897 * {@link TemporalAdjuster#adjustInto(Temporal)} method on the
898 * specified adjuster passing {@code this} as the argument.
899 * <p>
900 * This instance is immutable and unaffected by this method call.
901 *
902 * @param adjuster the adjuster to use, not null
903 * @return an {@code OffsetDateTime} based on {@code this} with the adjustment made, not null
904 * @throws DateTimeException if the adjustment cannot be made
905 * @throws ArithmeticException if numeric overflow occurs
906 */
907 @Override
908 public OffsetDateTime with(TemporalAdjuster adjuster) {
909     // optimizations
910     if (adjuster instanceof LocalDate || adjuster instanceof LocalTime || adjuster instanceof
        LocalDateTime) {
911         return with(dateTime.with(adjuster), offset);
912     } else if (adjuster instanceof Instant) {
913         return ofInstant((Instant) adjuster, offset);
914     } else if (adjuster instanceof ZoneOffset) {
915         return with(dateTime, (ZoneOffset) adjuster);
916     } else if (adjuster instanceof OffsetDateTime) {
917         return (OffsetDateTime) adjuster;
918     }
919     return (OffsetDateTime) adjuster.adjustInto(this);
920 }
921
922 /**
923 * Returns a copy of this date-time with the specified field set to a new value.
924 * <p>
925 * This returns an {@code OffsetDateTime}, based on this one, with the value
926 * for the specified field changed.
927 * This can be used to change any supported field, such as the year, month or day-of-month.
928 * If it is not possible to set the value, because the field is not supported or for
929 * some other reason, an exception is thrown.
930 * <p>
931 * In some cases, changing the specified field can cause the resulting date-time to become
        invalid,
932 * such as changing the month from 31st January to February would make the day-of-month invalid
        .
933 * In cases like this, the field is responsible for resolving the date. Typically it will

```

```

        choose
934     * the previous valid date, which would be the last valid day of February in this example.
935     * <p>
936     * If the field is a {@link ChronoField} then the adjustment is implemented here.
937     * <p>
938     * The {@code INSTANT_SECONDS} field will return a date-time with the specified instant.
939     * The offset and nano-of-second are unchanged.
940     * If the new instant value is outside the valid range then a {@code DateTimeException} will be
        thrown.
941     * <p>
942     * The {@code OFFSET_SECONDS} field will return a date-time with the specified offset.
943     * The local date-time is unaltered. If the new offset value is outside the valid range
944     * then a {@code DateTimeException} will be thrown.
945     * <p>
946     * The other {@link #isSupported(TemporalField) supported fields} will behave as per
947     * the matching method on {@link LocalDateTime#with(TemporalField, long) LocalDateTime}.
948     * In this case, the offset is not part of the calculation and will be unchanged.
949     * <p>
950     * All other {@code ChronoField} instances will throw an {@code
        UnsupportedTemporalTypeException}.
951     * <p>
952     * If the field is not a {@code ChronoField}, then the result of this method
953     * is obtained by invoking {@code TemporalField.adjustInto(Temporal, long)}
954     * passing {@code this} as the argument. In this case, the field determines
955     * whether and how to adjust the instant.
956     * <p>
957     * This instance is immutable and unaffected by this method call.
958     *
959     * @param field the field to set in the result, not null
960     * @param newValue the new value of the field in the result
961     * @return an {@code OffsetDateTime} based on {@code this} with the specified field set, not
        null
962     * @throws DateTimeException if the field cannot be set
963     * @throws UnsupportedTemporalTypeException if the field is not supported
964     * @throws ArithmeticException if numeric overflow occurs
965     */
966     @Override
967     public OffsetDateTime with(TemporalField field, long newValue) {
968         if (field instanceof ChronoField) {
969             ChronoField f = (ChronoField) field;
970             switch (f) {
971                 case INSTANT_SECONDS: return ofInstant(Instant.ofEpochSecond(newValue, getNano()),
                    offset);
972                 case OFFSET_SECONDS: {
973                     return with(dateTime, ZoneOffset.ofTotalSeconds(f.checkValidIntValue(newValue))
                    );
974                 }
975             }
976             return with(dateTime.with(field, newValue), offset);
977         }
978         return field.adjustInto(this, newValue);
979     }
980

```

```

981 //-----
982 /**
983  * Returns a copy of this {@code OffsetDateTime} with the year altered.
984  * <p>
985  * The time and offset do not affect the calculation and will be the same in the result.
986  * If the day-of-month is invalid for the year, it will be changed to the last valid day of the
987    month.
988  * <p>
989  * This instance is immutable and unaffected by this method call.
990  *
991  * @param year the year to set in the result, from MIN_YEAR to MAX_YEAR
992  * @return an {@code OffsetDateTime} based on this date-time with the requested year, not null
993  * @throws DateTimeException if the year value is invalid
994  */
995 public OffsetDateTime withYear(int year) {
996     return with(dateTime.withYear(year), offset);
997 }
998 /**
999  * Returns a copy of this {@code OffsetDateTime} with the month-of-year altered.
1000  * <p>
1001  * The time and offset do not affect the calculation and will be the same in the result.
1002  * If the day-of-month is invalid for the year, it will be changed to the last valid day of the
1003    month.
1004  * <p>
1005  * This instance is immutable and unaffected by this method call.
1006  *
1007  * @param month the month-of-year to set in the result, from 1 (January) to 12 (December)
1008  * @return an {@code OffsetDateTime} based on this date-time with the requested month, not null
1009  * @throws DateTimeException if the month-of-year value is invalid
1010  */
1011 public OffsetDateTime withMonth(int month) {
1012     return with(dateTime.withMonth(month), offset);
1013 }
1014 /**
1015  * Returns a copy of this {@code OffsetDateTime} with the day-of-month altered.
1016  * <p>
1017  * If the resulting {@code OffsetDateTime} is invalid, an exception is thrown.
1018  * The time and offset do not affect the calculation and will be the same in the result.
1019  * <p>
1020  * This instance is immutable and unaffected by this method call.
1021  *
1022  * @param dayOfMonth the day-of-month to set in the result, from 1 to 28-31
1023  * @return an {@code OffsetDateTime} based on this date-time with the requested day, not null
1024  * @throws DateTimeException if the day-of-month value is invalid,
1025  *       or if the day-of-month is invalid for the month-year
1026  */
1027 public OffsetDateTime withDayOfMonth(int dayOfMonth) {
1028     return with(dateTime.withDayOfMonth(dayOfMonth), offset);
1029 }
1030
1031 /**

```

```

1032     * Returns a copy of this {@code OffsetDateTime} with the day-of-year altered.
1033     * <p>
1034     * The time and offset do not affect the calculation and will be the same in the result.
1035     * If the resulting {@code OffsetDateTime} is invalid, an exception is thrown.
1036     * <p>
1037     * This instance is immutable and unaffected by this method call.
1038     *
1039     * @param dayOfYear the day-of-year to set in the result, from 1 to 365-366
1040     * @return an {@code OffsetDateTime} based on this date with the requested day, not null
1041     * @throws DateTimeException if the day-of-year value is invalid,
1042     *         or if the day-of-year is invalid for the year
1043     */
1044     public OffsetDateTime withDayOfYear(int dayOfYear) {
1045         return with(dateTime.withDayOfYear(dayOfYear), offset);
1046     }
1047
1048     //-----
1049     /**
1050     * Returns a copy of this {@code OffsetDateTime} with the hour-of-day altered.
1051     * <p>
1052     * The date and offset do not affect the calculation and will be the same in the result.
1053     * <p>
1054     * This instance is immutable and unaffected by this method call.
1055     *
1056     * @param hour the hour-of-day to set in the result, from 0 to 23
1057     * @return an {@code OffsetDateTime} based on this date-time with the requested hour, not null
1058     * @throws DateTimeException if the hour value is invalid
1059     */
1060     public OffsetDateTime withHour(int hour) {
1061         return with(dateTime.withHour(hour), offset);
1062     }
1063
1064     /**
1065     * Returns a copy of this {@code OffsetDateTime} with the minute-of-hour altered.
1066     * <p>
1067     * The date and offset do not affect the calculation and will be the same in the result.
1068     * <p>
1069     * This instance is immutable and unaffected by this method call.
1070     *
1071     * @param minute the minute-of-hour to set in the result, from 0 to 59
1072     * @return an {@code OffsetDateTime} based on this date-time with the requested minute, not
1073         null
1074     * @throws DateTimeException if the minute value is invalid
1075     */
1076     public OffsetDateTime withMinute(int minute) {
1077         return with(dateTime.withMinute(minute), offset);
1078     }
1079
1080     /**
1081     * Returns a copy of this {@code OffsetDateTime} with the second-of-minute altered.
1082     * <p>
1083     * The date and offset do not affect the calculation and will be the same in the result.
1084     * <p>

```

```

1084     * This instance is immutable and unaffected by this method call.
1085     *
1086     * @param second the second-of-minute to set in the result, from 0 to 59
1087     * @return an {@code OffsetDateTime} based on this date-time with the requested second, not
1088     *         null
1089     * @throws DateTimeException if the second value is invalid
1090     */
1091     public OffsetDateTime withSecond(int second) {
1092         return with(dateTime.withSecond(second), offset);
1093     }
1094
1095     /**
1096     * Returns a copy of this {@code OffsetDateTime} with the nano-of-second altered.
1097     * <p>
1098     * The date and offset do not affect the calculation and will be the same in the result.
1099     * <p>
1100     * This instance is immutable and unaffected by this method call.
1101     *
1102     * @param nanoOfSecond the nano-of-second to set in the result, from 0 to 999,999,999
1103     * @return an {@code OffsetDateTime} based on this date-time with the requested nanosecond, not
1104     *         null
1105     * @throws DateTimeException if the nano value is invalid
1106     */
1107     public OffsetDateTime withNano(int nanoOfSecond) {
1108         return with(dateTime.withNano(nanoOfSecond), offset);
1109     }
1110
1111     //-----
1112     /**
1113     * Returns a copy of this {@code OffsetDateTime} with the time truncated.
1114     * <p>
1115     * Truncation returns a copy of the original date-time with fields
1116     * smaller than the specified unit set to zero.
1117     * For example, truncating with the {@link ChronoUnit#MINUTES minutes} unit
1118     * will set the second-of-minute and nano-of-second field to zero.
1119     * <p>
1120     * The unit must have a {@link TemporalUnit#getDuration() duration}
1121     * that divides into the length of a standard day without remainder.
1122     * This includes all supplied time units on {@link ChronoUnit} and
1123     * {@link ChronoUnit#DAYS DAYS}. Other units throw an exception.
1124     * <p>
1125     * The offset does not affect the calculation and will be the same in the result.
1126     * <p>
1127     * This instance is immutable and unaffected by this method call.
1128     *
1129     * @param unit the unit to truncate to, not null
1130     * @return an {@code OffsetDateTime} based on this date-time with the time truncated, not null
1131     * @throws DateTimeException if unable to truncate
1132     * @throws UnsupportedTemporalTypeException if the unit is not supported
1133     */
1134     public OffsetDateTime truncatedTo(TemporalUnit unit) {
1135         return with(dateTime.truncatedTo(unit), offset);
1136     }

```

```

1135 //-----
1136 /**
1137  * Returns a copy of this date-time with the specified amount added.
1138  * <p>
1139  * This returns an {@code OffsetDateTime}, based on this one, with the specified amount added.
1140  * The amount is typically {@link Period} or {@link Duration} but may be
1141  * any other type implementing the {@link TemporalAmount} interface.
1142  * <p>
1143  * The calculation is delegated to the amount object by calling
1144  * {@link TemporalAmount#addTo(Temporal)}. The amount implementation is free
1145  * to implement the addition in any way it wishes, however it typically
1146  * calls back to {@link #plus(long, TemporalUnit)}. Consult the documentation
1147  * of the amount implementation to determine if it can be successfully added.
1148  * <p>
1149  * This instance is immutable and unaffected by this method call.
1150  *
1151  * @param amountToAdd the amount to add, not null
1152  * @return an {@code OffsetDateTime} based on this date-time with the addition made, not null
1153  * @throws DateTimeException if the addition cannot be made
1154  * @throws ArithmeticException if numeric overflow occurs
1155  */
1156 @Override
1157 public OffsetDateTime plus(TemporalAmount amountToAdd) {
1158     return (OffsetDateTime) amountToAdd.addTo(this);
1159 }
1160
1161 /**
1162  * Returns a copy of this date-time with the specified amount added.
1163  * <p>
1164  * This returns an {@code OffsetDateTime}, based on this one, with the amount
1165  * in terms of the unit added. If it is not possible to add the amount, because the
1166  * unit is not supported or for some other reason, an exception is thrown.
1167  * <p>
1168  * If the field is a {@link ChronoUnit} then the addition is implemented by
1169  * {@link LocalDateTime#plus(long, TemporalUnit)}.
1170  * The offset is not part of the calculation and will be unchanged in the result.
1171  * <p>
1172  * If the field is not a {@code ChronoUnit}, then the result of this method
1173  * is obtained by invoking {@code TemporalUnit.addTo(Temporal, long)}
1174  * passing {@code this} as the argument. In this case, the unit determines
1175  * whether and how to perform the addition.
1176  * <p>
1177  * This instance is immutable and unaffected by this method call.
1178  *
1179  * @param amountToAdd the amount of the unit to add to the result, may be negative
1180  * @param unit the unit of the amount to add, not null
1181  * @return an {@code OffsetDateTime} based on this date-time with the specified amount added,
1182  *         not null
1183  * @throws DateTimeException if the addition cannot be made
1184  * @throws UnsupportedTemporalTypeException if the unit is not supported
1185  * @throws ArithmeticException if numeric overflow occurs
1186  */

```



```

1187 @Override
1188 public OffsetDateTime plus(long amountToAdd, TemporalUnit unit) {
1189     if (unit instanceof ChronoUnit) {
1190         return with(dateTime.plus(amountToAdd, unit), offset);
1191     }
1192     return unit.addTo(this, amountToAdd);
1193 }
1194
1195 //-----
1196 /**
1197  * Returns a copy of this {@code OffsetDateTime} with the specified number of years added.
1198  * <p>
1199  * This method adds the specified amount to the years field in three steps:
1200  * <ol>
1201  * <li>Add the input years to the year field</li>
1202  * <li>Check if the resulting date would be invalid</li>
1203  * <li>Adjust the day-of-month to the last valid day if necessary</li>
1204  * </ol>
1205  * <p>
1206  * For example, 2008-02-29 (leap year) plus one year would result in the
1207  * invalid date 2009-02-29 (standard year). Instead of returning an invalid
1208  * result, the last valid day of the month, 2009-02-28, is selected instead.
1209  * <p>
1210  * This instance is immutable and unaffected by this method call.
1211  *
1212  * @param years the years to add, may be negative
1213  * @return an {@code OffsetDateTime} based on this date-time with the years added, not null
1214  * @throws DateTimeException if the result exceeds the supported date range
1215  */
1216 public OffsetDateTime plusYears(long years) {
1217     return with(dateTime.plusYears(years), offset);
1218 }
1219
1220 /**
1221  * Returns a copy of this {@code OffsetDateTime} with the specified number of months added.
1222  * <p>
1223  * This method adds the specified amount to the months field in three steps:
1224  * <ol>
1225  * <li>Add the input months to the month-of-year field</li>
1226  * <li>Check if the resulting date would be invalid</li>
1227  * <li>Adjust the day-of-month to the last valid day if necessary</li>
1228  * </ol>
1229  * <p>
1230  * For example, 2007-03-31 plus one month would result in the invalid date
1231  * 2007-04-31. Instead of returning an invalid result, the last valid day
1232  * of the month, 2007-04-30, is selected instead.
1233  * <p>
1234  * This instance is immutable and unaffected by this method call.
1235  *
1236  * @param months the months to add, may be negative
1237  * @return an {@code OffsetDateTime} based on this date-time with the months added, not null
1238  * @throws DateTimeException if the result exceeds the supported date range
1239  */

```

```
1240 public OffsetDateTime plusMonths(long months) {
1241     return with(dateTime.plusMonths(months), offset);
1242 }
1243
1244 /**
1245  * Returns a copy of this OffsetDateTime with the specified number of weeks added.
1246  * <p>
1247  * This method adds the specified amount in weeks to the days field incrementing
1248  * the month and year fields as necessary to ensure the result remains valid.
1249  * The result is only invalid if the maximum/minimum year is exceeded.
1250  * <p>
1251  * For example, 2008-12-31 plus one week would result in 2009-01-07.
1252  * <p>
1253  * This instance is immutable and unaffected by this method call.
1254  *
1255  * @param weeks the weeks to add, may be negative
1256  * @return an {@code OffsetDateTime} based on this date-time with the weeks added, not null
1257  * @throws DateTimeException if the result exceeds the supported date range
1258  */
1259 public OffsetDateTime plusWeeks(long weeks) {
1260     return with(dateTime.plusWeeks(weeks), offset);
1261 }
1262
1263 /**
1264  * Returns a copy of this OffsetDateTime with the specified number of days added.
1265  * <p>
1266  * This method adds the specified amount to the days field incrementing the
1267  * month and year fields as necessary to ensure the result remains valid.
1268  * The result is only invalid if the maximum/minimum year is exceeded.
1269  * <p>
1270  * For example, 2008-12-31 plus one day would result in 2009-01-01.
1271  * <p>
1272  * This instance is immutable and unaffected by this method call.
1273  *
1274  * @param days the days to add, may be negative
1275  * @return an {@code OffsetDateTime} based on this date-time with the days added, not null
1276  * @throws DateTimeException if the result exceeds the supported date range
1277  */
1278 public OffsetDateTime plusDays(long days) {
1279     return with(dateTime.plusDays(days), offset);
1280 }
1281
1282 /**
1283  * Returns a copy of this {@code OffsetDateTime} with the specified number of hours added.
1284  * <p>
1285  * This instance is immutable and unaffected by this method call.
1286  *
1287  * @param hours the hours to add, may be negative
1288  * @return an {@code OffsetDateTime} based on this date-time with the hours added, not null
1289  * @throws DateTimeException if the result exceeds the supported date range
1290  */
1291 public OffsetDateTime plusHours(long hours) {
1292     return with(dateTime.plusHours(hours), offset);
1293 }
```

```

1293     }
1294
1295     /**
1296      * Returns a copy of this {@code OffsetDateTime} with the specified number of minutes added.
1297      * <p>
1298      * This instance is immutable and unaffected by this method call.
1299      *
1300      * @param minutes the minutes to add, may be negative
1301      * @return an {@code OffsetDateTime} based on this date-time with the minutes added, not null
1302      * @throws DateTimeException if the result exceeds the supported date range
1303      */
1304     public OffsetDateTime plusMinutes(long minutes) {
1305         return with(dateTime.plusMinutes(minutes), offset);
1306     }
1307
1308     /**
1309      * Returns a copy of this {@code OffsetDateTime} with the specified number of seconds added.
1310      * <p>
1311      * This instance is immutable and unaffected by this method call.
1312      *
1313      * @param seconds the seconds to add, may be negative
1314      * @return an {@code OffsetDateTime} based on this date-time with the seconds added, not null
1315      * @throws DateTimeException if the result exceeds the supported date range
1316      */
1317     public OffsetDateTime plusSeconds(long seconds) {
1318         return with(dateTime.plusSeconds(seconds), offset);
1319     }
1320
1321     /**
1322      * Returns a copy of this {@code OffsetDateTime} with the specified number of nanoseconds added
1323      *
1324      * <p>
1325      * This instance is immutable and unaffected by this method call.
1326      *
1327      * @param nanos the nanos to add, may be negative
1328      * @return an {@code OffsetDateTime} based on this date-time with the nanoseconds added, not
1329      * null
1330      * @throws DateTimeException if the unit cannot be added to this type
1331      */
1332     public OffsetDateTime plusNanos(long nanos) {
1333         return with(dateTime.plusNanos(nanos), offset);
1334     }
1335
1336     //-----
1337     /**
1338      * Returns a copy of this date-time with the specified amount subtracted.
1339      * <p>
1340      * This returns an {@code OffsetDateTime}, based on this one, with the specified amount
1341      * subtracted.
1342      *
1343      * The amount is typically {@link Period} or {@link Duration} but may be
1344      * any other type implementing the {@link TemporalAmount} interface.
1345      * <p>
1346      * The calculation is delegated to the amount object by calling

```

```

1343 * {@link TemporalAmount#subtractFrom(Temporal)}. The amount implementation is free
1344 * to implement the subtraction in any way it wishes, however it typically
1345 * calls back to {@link #minus(long, TemporalUnit)}. Consult the documentation
1346 * of the amount implementation to determine if it can be successfully subtracted.
1347 * <p>
1348 * This instance is immutable and unaffected by this method call.
1349 *
1350 * @param amountToSubtract the amount to subtract, not null
1351 * @return an {@code OffsetDateTime} based on this date-time with the subtraction made, not
        null
1352 * @throws DateTimeException if the subtraction cannot be made
1353 * @throws ArithmeticException if numeric overflow occurs
1354 */
1355 @Override
1356 public OffsetDateTime minus(TemporalAmount amountToSubtract) {
1357     return (OffsetDateTime) amountToSubtract.subtractFrom(this);
1358 }
1359
1360 /**
1361 * Returns a copy of this date-time with the specified amount subtracted.
1362 * <p>
1363 * This returns an {@code OffsetDateTime}, based on this one, with the amount
1364 * in terms of the unit subtracted. If it is not possible to subtract the amount,
1365 * because the unit is not supported or for some other reason, an exception is thrown.
1366 * <p>
1367 * This method is equivalent to {@link #plus(long, TemporalUnit)} with the amount negated.
1368 * See that method for a full description of how addition, and thus subtraction, works.
1369 * <p>
1370 * This instance is immutable and unaffected by this method call.
1371 *
1372 * @param amountToSubtract the amount of the unit to subtract from the result, may be negative
1373 * @param unit the unit of the amount to subtract, not null
1374 * @return an {@code OffsetDateTime} based on this date-time with the specified amount
        subtracted, not null
1375 * @throws DateTimeException if the subtraction cannot be made
1376 * @throws UnsupportedTemporalTypeException if the unit is not supported
1377 * @throws ArithmeticException if numeric overflow occurs
1378 */
1379 @Override
1380 public OffsetDateTime minus(long amountToSubtract, TemporalUnit unit) {
1381     return (amountToSubtract == Long.MIN_VALUE ? plus(Long.MAX_VALUE, unit).plus(1, unit) :
        plus(-amountToSubtract, unit));
1382 }
1383
1384 //-----
1385 /**
1386 * Returns a copy of this {@code OffsetDateTime} with the specified number of years subtracted.
1387 * <p>
1388 * This method subtracts the specified amount from the years field in three steps:
1389 * <ol>
1390 * <li>Subtract the input years from the year field</li>
1391 * <li>Check if the resulting date would be invalid</li>
1392 * <li>Adjust the day-of-month to the last valid day if necessary</li>

```

```

1393     * </ol>
1394     * <p>
1395     * For example, 2008-02-29 (leap year) minus one year would result in the
1396     * invalid date 2009-02-29 (standard year). Instead of returning an invalid
1397     * result, the last valid day of the month, 2009-02-28, is selected instead.
1398     * <p>
1399     * This instance is immutable and unaffected by this method call.
1400     *
1401     * @param years the years to subtract, may be negative
1402     * @return an {@code OffsetDateTime} based on this date-time with the years subtracted, not
1403     *         null
1404     * @throws DateTimeException if the result exceeds the supported date range
1405     */
1406     public OffsetDateTime minusYears(long years) {
1407         return (years == Long.MIN_VALUE ? plusYears(Long.MAX_VALUE).plusYears(1) : plusYears(-years
1408         ));
1409     }
1410
1411     /**
1412     * Returns a copy of this {@code OffsetDateTime} with the specified number of months subtracted
1413     * .
1414     * <p>
1415     * This method subtracts the specified amount from the months field in three steps:
1416     * <ol>
1417     * <li>Subtract the input months from the month-of-year field</li>
1418     * <li>Check if the resulting date would be invalid</li>
1419     * <li>Adjust the day-of-month to the last valid day if necessary</li>
1420     * </ol>
1421     * <p>
1422     * For example, 2007-03-31 minus one month would result in the invalid date
1423     * 2007-04-31. Instead of returning an invalid result, the last valid day
1424     * of the month, 2007-04-30, is selected instead.
1425     * <p>
1426     * This instance is immutable and unaffected by this method call.
1427     *
1428     * @param months the months to subtract, may be negative
1429     * @return an {@code OffsetDateTime} based on this date-time with the months subtracted, not
1430     *         null
1431     * @throws DateTimeException if the result exceeds the supported date range
1432     */
1433     public OffsetDateTime minusMonths(long months) {
1434         return (months == Long.MIN_VALUE ? plusMonths(Long.MAX_VALUE).plusMonths(1) : plusMonths(-
1435         months));
1436     }
1437
1438     /**
1439     * Returns a copy of this {@code OffsetDateTime} with the specified number of weeks subtracted.
1440     * <p>
1441     * This method subtracts the specified amount in weeks from the days field decrementing
1442     * the month and year fields as necessary to ensure the result remains valid.
1443     * The result is only invalid if the maximum/minimum year is exceeded.
1444     * <p>
1445     * For example, 2008-12-31 minus one week would result in 2009-01-07.

```

```

1441     * <p>
1442     * This instance is immutable and unaffected by this method call.
1443     *
1444     * @param weeks  the weeks to subtract, may be negative
1445     * @return an {@code OffsetDateTime} based on this date-time with the weeks subtracted, not
1446     *         null
1447     * @throws DateTimeException if the result exceeds the supported date range
1448     */
1449     public OffsetDateTime minusWeeks(long weeks) {
1450         return (weeks == Long.MIN_VALUE ? plusWeeks(Long.MAX_VALUE).plusWeeks(1) : plusWeeks(-weeks
1451         ));
1452     }
1453
1454     /**
1455     * Returns a copy of this {@code OffsetDateTime} with the specified number of days subtracted.
1456     * <p>
1457     * This method subtracts the specified amount from the days field decrementing the
1458     * month and year fields as necessary to ensure the result remains valid.
1459     * The result is only invalid if the maximum/minimum year is exceeded.
1460     * <p>
1461     * For example, 2008-12-31 minus one day would result in 2009-01-01.
1462     * <p>
1463     * This instance is immutable and unaffected by this method call.
1464     *
1465     * @param days  the days to subtract, may be negative
1466     * @return an {@code OffsetDateTime} based on this date-time with the days subtracted, not null
1467     * @throws DateTimeException if the result exceeds the supported date range
1468     */
1469     public OffsetDateTime minusDays(long days) {
1470         return (days == Long.MIN_VALUE ? plusDays(Long.MAX_VALUE).plusDays(1) : plusDays(-days));
1471     }
1472
1473     /**
1474     * Returns a copy of this {@code OffsetDateTime} with the specified number of hours subtracted.
1475     * <p>
1476     * This instance is immutable and unaffected by this method call.
1477     *
1478     * @param hours  the hours to subtract, may be negative
1479     * @return an {@code OffsetDateTime} based on this date-time with the hours subtracted, not
1480     *         null
1481     * @throws DateTimeException if the result exceeds the supported date range
1482     */
1483     public OffsetDateTime minusHours(long hours) {
1484         return (hours == Long.MIN_VALUE ? plusHours(Long.MAX_VALUE).plusHours(1) : plusHours(-hours
1485         ));
1486     }
1487
1488     /**
1489     * Returns a copy of this {@code OffsetDateTime} with the specified number of minutes
1490     * subtracted.
1491     * <p>
1492     * This instance is immutable and unaffected by this method call.
1493     *

```

```

1489     * @param minutes the minutes to subtract, may be negative
1490     * @return an {@code OffsetDateTime} based on this date-time with the minutes subtracted, not
1491     *         null
1492     * @throws DateTimeException if the result exceeds the supported date range
1493     */
1494     public OffsetDateTime minusMinutes(long minutes) {
1495         return (minutes == Long.MIN_VALUE ? plusMinutes(Long.MAX_VALUE).plusMinutes(1) :
1496             plusMinutes(-minutes));
1497     }
1498
1499     /**
1500     * Returns a copy of this {@code OffsetDateTime} with the specified number of seconds
1501     * subtracted.
1502     * <p>
1503     * This instance is immutable and unaffected by this method call.
1504     * </p>
1505     * @param seconds the seconds to subtract, may be negative
1506     * @return an {@code OffsetDateTime} based on this date-time with the seconds subtracted, not
1507     *         null
1508     * @throws DateTimeException if the result exceeds the supported date range
1509     */
1510     public OffsetDateTime minusSeconds(long seconds) {
1511         return (seconds == Long.MIN_VALUE ? plusSeconds(Long.MAX_VALUE).plusSeconds(1) :
1512             plusSeconds(-seconds));
1513     }
1514
1515     /**
1516     * Returns a copy of this {@code OffsetDateTime} with the specified number of nanoseconds
1517     * subtracted.
1518     * <p>
1519     * This instance is immutable and unaffected by this method call.
1520     * </p>
1521     * @param nanos the nanos to subtract, may be negative
1522     * @return an {@code OffsetDateTime} based on this date-time with the nanoseconds subtracted,
1523     *         not null
1524     * @throws DateTimeException if the result exceeds the supported date range
1525     */
1526     public OffsetDateTime minusNanos(long nanos) {
1527         return (nanos == Long.MIN_VALUE ? plusNanos(Long.MAX_VALUE).plusNanos(1) : plusNanos(-nanos
1528             ));
1529     }
1530
1531     //-----
1532     /**
1533     * Queries this date-time using the specified query.
1534     * <p>
1535     * This queries this date-time using the specified query strategy object.
1536     * The {@code TemporalQuery} object defines the logic to be used to
1537     * obtain the result. Read the documentation of the query to understand
1538     * what the result of this method will be.
1539     * </p>
1540     * The result of this method is obtained by invoking the
1541     * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the

```

```

1534     * specified query passing {@code this} as the argument.
1535     *
1536     * @param <R> the type of the result
1537     * @param query the query to invoke, not null
1538     * @return the query result, null may be returned (defined by the query)
1539     * @throws DateTimeException if unable to query (defined by the query)
1540     * @throws ArithmeticException if numeric overflow occurs (defined by the query)
1541     */
1542     @SuppressWarnings("unchecked")
1543     @Override
1544     public <R> R query(TemporalQuery<R> query) {
1545         if (query == TemporalQueries.offset() || query == TemporalQueries.zone()) {
1546             return (R) getOffset();
1547         } else if (query == TemporalQueries.zoneId()) {
1548             return null;
1549         } else if (query == TemporalQueries.localDate()) {
1550             return (R) toLocalDate();
1551         } else if (query == TemporalQueries.localTime()) {
1552             return (R) toLocalTime();
1553         } else if (query == TemporalQueries.chronology()) {
1554             return (R) IsoChronology.INSTANCE;
1555         } else if (query == TemporalQueries.precision()) {
1556             return (R) NANOS;
1557         }
1558         // inline TemporalAccessor.super.query(query) as an optimization
1559         // non-JDK classes are not permitted to make this optimization
1560         return query.queryFrom(this);
1561     }
1562
1563     /**
1564     * Adjusts the specified temporal object to have the same offset, date
1565     * and time as this object.
1566     * <p>
1567     * This returns a temporal object of the same observable type as the input
1568     * with the offset, date and time changed to be the same as this.
1569     * <p>
1570     * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
1571     * three times, passing {@link ChronoField#EPOCH_DAY},
1572     * {@link ChronoField#NANO_OF_DAY} and {@link ChronoField#OFFSET_SECONDS} as the fields.
1573     * <p>
1574     * In most cases, it is clearer to reverse the calling pattern by using
1575     * {@link Temporal#with(TemporalAdjuster)}:
1576     * <pre>
1577     * // these two lines are equivalent, but the second approach is recommended
1578     * temporal = thisOffsetDateTime.adjustInto(temporal);
1579     * temporal = temporal.with(thisOffsetDateTime);
1580     * </pre>
1581     * <p>
1582     * This instance is immutable and unaffected by this method call.
1583     *
1584     * @param temporal the target object to be adjusted, not null
1585     * @return the adjusted object, not null
1586     * @throws DateTimeException if unable to make the adjustment

```



```

1587     * @throws ArithmeticException if numeric overflow occurs
1588     */
1589     @Override
1590     public Temporal adjustInto(Temporal temporal) {
1591         // OffsetDateTime is treated as three separate fields, not an instant
1592         // this produces the most consistent set of results overall
1593         // the offset is set after the date and time, as it is typically a small
1594         // tweak to the result, with ZonedDateTime frequently ignoring the offset
1595         return temporal
1596             .with(EPOCH_DAY, toLocalDate().toEpochDay())
1597             .with(NANO_OF_DAY, toLocalTime().toNanoOfDay())
1598             .with(OFFSET_SECONDS, getOffset().getTotalSeconds());
1599     }
1600
1601     /**
1602     * Calculates the amount of time until another date-time in terms of the specified unit.
1603     * <p>
1604     * This calculates the amount of time between two {@code OffsetDateTime}
1605     * objects in terms of a single {@code TemporalUnit}.
1606     * The start and end points are {@code this} and the specified date-time.
1607     * The result will be negative if the end is before the start.
1608     * For example, the amount in days between two date-times can be calculated
1609     * using {@code startDateTime.until(endDateTime, DAYS)}.
1610     * <p>
1611     * The {@code Temporal} passed to this method is converted to a
1612     * {@code OffsetDateTime} using {@code #from(TemporalAccessor)}.
1613     * If the offset differs between the two date-times, the specified
1614     * end date-time is normalized to have the same offset as this date-time.
1615     * <p>
1616     * The calculation returns a whole number, representing the number of
1617     * complete units between the two date-times.
1618     * For example, the amount in months between 2012-06-15T00:00Z and 2012-08-14T23:59Z
1619     * will only be one month as it is one minute short of two months.
1620     * <p>
1621     * There are two equivalent ways of using this method.
1622     * The first is to invoke this method.
1623     * The second is to use {@code TemporalUnit#between(Temporal, Temporal)}:
1624     * <pre>
1625     *     // these two lines are equivalent
1626     *     amount = start.until(end, MONTHS);
1627     *     amount = MONTHS.between(start, end);
1628     * </pre>
1629     * The choice should be made based on which makes the code more readable.
1630     * <p>
1631     * The calculation is implemented in this method for {@code ChronoUnit}.
1632     * The units {@code NANOS}, {@code MICROS}, {@code MILLIS}, {@code SECONDS},
1633     * {@code MINUTES}, {@code HOURS} and {@code HALF_DAYS}, {@code DAYS},
1634     * {@code WEEKS}, {@code MONTHS}, {@code YEARS}, {@code DECADES},
1635     * {@code CENTURIES}, {@code MILLENNIA} and {@code ERAS} are supported.
1636     * Other {@code ChronoUnit} values will throw an exception.
1637     * <p>
1638     * If the unit is not a {@code ChronoUnit}, then the result of this method
1639     * is obtained by invoking {@code TemporalUnit.between(Temporal, Temporal)}

```

```

1640     * passing {@code this} as the first argument and the converted input temporal
1641     * as the second argument.
1642     * <p>
1643     * This instance is immutable and unaffected by this method call.
1644     *
1645     * @param endExclusive the end date, exclusive, which is converted to an {@code OffsetDateTime}
1646         }, not null
1647     * @param unit the unit to measure the amount in, not null
1648     * @return the amount of time between this date-time and the end date-time
1649     * @throws DateTimeException if the amount cannot be calculated, or the end
1650     *     temporal cannot be converted to an {@code OffsetDateTime}
1651     * @throws UnsupportedTemporalTypeException if the unit is not supported
1652     * @throws ArithmeticException if numeric overflow occurs
1653     */
1654     @Override
1655     public long until(Temporal endExclusive, TemporalUnit unit) {
1656         OffsetDateTime end = OffsetDateTime.from(endExclusive);
1657         if (unit instanceof ChronoUnit) {
1658             end = end.withOffsetSameInstant(offset);
1659             return dateTime.until(end.dateTime, unit);
1660         }
1661         return unit.between(this, end);
1662     }
1663
1664     /**
1665     * Formats this date-time using the specified formatter.
1666     * <p>
1667     * This date-time will be passed to the formatter to produce a string.
1668     *
1669     * @param formatter the formatter to use, not null
1670     * @return the formatted date-time string, not null
1671     * @throws DateTimeException if an error occurs during printing
1672     */
1673     public String format(DateTimeFormatter formatter) {
1674         Objects.requireNonNull(formatter, "formatter");
1675         return formatter.format(this);
1676     }
1677
1678     //-----
1679     /**
1680     * Combines this date-time with a time-zone to create a {@code ZonedDateTime}
1681     * ensuring that the result has the same instant.
1682     * <p>
1683     * This returns a {@code ZonedDateTime} formed from this date-time and the specified time-zone.
1684     * This conversion will ignore the visible local date-time and use the underlying instant
1685     * instead.
1686     * This avoids any problems with local time-line gaps or overlaps.
1687     * The result might have different values for fields such as hour, minute and even day.
1688     * <p>
1689     * To attempt to retain the values of the fields, use {@link #atZoneSimilarLocal(ZoneId)}.
1690     * To use the offset as the zone ID, use {@link #toZonedDateTime()}.
1691     *
1692     * @param zone the time-zone to use, not null

```

```

1691     * @return the zoned date-time formed from this date-time, not null
1692     */
1693     public ZonedDateTime atZoneSameInstant(ZoneId zone) {
1694         return ZonedDateTime.ofInstant(dateTime, offset, zone);
1695     }
1696
1697     /**
1698     * Combines this date-time with a time-zone to create a {@code ZonedDateTime}
1699     * trying to keep the same local date and time.
1700     * <p>
1701     * This returns a {@code ZonedDateTime} formed from this date-time and the specified time-zone.
1702     * Where possible, the result will have the same local date-time as this object.
1703     * <p>
1704     * Time-zone rules, such as daylight savings, mean that not every time on the
1705     * local time-line exists. If the local date-time is in a gap or overlap according to
1706     * the rules then a resolver is used to determine the resultant local time and offset.
1707     * This method uses {@link ZonedDateTime#ofLocal(LocalDateTime, ZoneId, ZoneOffset)}
1708     * to retain the offset from this instance if possible.
1709     * <p>
1710     * Finer control over gaps and overlaps is available in two ways.
1711     * If you simply want to use the later offset at overlaps then call
1712     * {@link ZonedDateTime#withLaterOffsetAtOverlap()} immediately after this method.
1713     * <p>
1714     * To create a zoned date-time at the same instant irrespective of the local time-line,
1715     * use {@link #atZoneSameInstant(ZoneId)}.
1716     * To use the offset as the zone ID, use {@link #toZonedDateTime()}.
1717     *
1718     * @param zone the time-zone to use, not null
1719     * @return the zoned date-time formed from this date and the earliest valid time for the zone,
1720     *         not null
1721     */
1722     public ZonedDateTime atZoneSimilarLocal(ZoneId zone) {
1723         return ZonedDateTime.ofLocal(dateTime, zone, offset);
1724     }
1725
1726     //-----
1727     /**
1728     * Converts this date-time to an {@code OffsetTime}.
1729     * <p>
1730     * This returns an offset time with the same local time and offset.
1731     *
1732     * @return an OffsetTime representing the time and offset, not null
1733     */
1734     public OffsetTime toOffsetTime() {
1735         return OffsetTime.of(dateTime.toLocalTime(), offset);
1736     }
1737
1738     /**
1739     * Converts this date-time to a {@code ZonedDateTime} using the offset as the zone ID.
1740     * <p>
1741     * This creates the simplest possible {@code ZonedDateTime} using the offset
1742     * as the zone ID.
1743     * <p>

```

```

1743     * To control the time-zone used, see {@link #atZoneSameInstant(ZoneId)} and
1744     * {@link #atZoneSimilarLocal(ZoneId)}.
1745     *
1746     * @return a zoned date-time representing the same local date-time and offset, not null
1747     */
1748     public ZonedDateTime toZonedDateTime() {
1749         return ZonedDateTime.of(dateTime, offset);
1750     }
1751
1752     /**
1753     * Converts this date-time to an {@code Instant}.
1754     * <p>
1755     * This returns an {@code Instant} representing the same point on the
1756     * time-line as this date-time.
1757     *
1758     * @return an {@code Instant} representing the same instant, not null
1759     */
1760     public Instant toInstant() {
1761         return dateTime.toInstant(offset);
1762     }
1763
1764     /**
1765     * Converts this date-time to the number of seconds from the epoch of 1970-01-01T00:00:00Z.
1766     * <p>
1767     * This allows this date-time to be converted to a value of the
1768     * {@link ChronoField#INSTANT_SECONDS epoch-seconds} field. This is primarily
1769     * intended for low-level conversions rather than general application usage.
1770     *
1771     * @return the number of seconds from the epoch of 1970-01-01T00:00:00Z
1772     */
1773     public long toEpochSecond() {
1774         return dateTime.toEpochSecond(offset);
1775     }
1776
1777     //-----
1778     /**
1779     * Compares this date-time to another date-time.
1780     * <p>
1781     * The comparison is based on the instant then on the local date-time.
1782     * It is "consistent with equals", as defined by {@link Comparable}.
1783     * <p>
1784     * For example, the following is the comparator order:
1785     * <ol>
1786     * <li>{@code 2008-12-03T10:30+01:00}</li>
1787     * <li>{@code 2008-12-03T11:00+01:00}</li>
1788     * <li>{@code 2008-12-03T12:00+02:00}</li>
1789     * <li>{@code 2008-12-03T11:30+01:00}</li>
1790     * <li>{@code 2008-12-03T12:00+01:00}</li>
1791     * <li>{@code 2008-12-03T12:30+01:00}</li>
1792     * </ol>
1793     * Values #2 and #3 represent the same instant on the time-line.
1794     * When two values represent the same instant, the local date-time is compared
1795     * to distinguish them. This step is needed to make the ordering

```

```

1796     * consistent with {@code equals()}.
1797     *
1798     * @param other the other date-time to compare to, not null
1799     * @return the comparator value, negative if less, positive if greater
1800     */
1801     @Override
1802     public int compareTo(OffsetDateTime other) {
1803         int cmp = compareInstant(this, other);
1804         if (cmp == 0) {
1805             cmp = toLocalDateTime().compareTo(other.toLocalDateTime());
1806         }
1807         return cmp;
1808     }
1809
1810     //-----
1811     /**
1812      * Checks if the instant of this date-time is after that of the specified date-time.
1813      * <p>
1814      * This method differs from the comparison in {@link #compareTo} and {@link #equals} in that it
1815      * only compares the instant of the date-time. This is equivalent to using
1816      * {@code dateTime1.toInstant().isAfter(dateTime2.toInstant())}.
1817      *
1818      * @param other the other date-time to compare to, not null
1819      * @return true if this is after the instant of the specified date-time
1820      */
1821     public boolean isAfter(OffsetDateTime other) {
1822         long thisEpochSec = toEpochSecond();
1823         long otherEpochSec = other.toEpochSecond();
1824         return thisEpochSec > otherEpochSec ||
1825             (thisEpochSec == otherEpochSec && toLocalTime().getNano() > other.toLocalTime().getNano());
1826     }
1827
1828     /**
1829      * Checks if the instant of this date-time is before that of the specified date-time.
1830      * <p>
1831      * This method differs from the comparison in {@link #compareTo} in that it
1832      * only compares the instant of the date-time. This is equivalent to using
1833      * {@code dateTime1.toInstant().isBefore(dateTime2.toInstant())}.
1834      *
1835      * @param other the other date-time to compare to, not null
1836      * @return true if this is before the instant of the specified date-time
1837      */
1838     public boolean isBefore(OffsetDateTime other) {
1839         long thisEpochSec = toEpochSecond();
1840         long otherEpochSec = other.toEpochSecond();
1841         return thisEpochSec < otherEpochSec ||
1842             (thisEpochSec == otherEpochSec && toLocalTime().getNano() < other.toLocalTime().getNano());
1843     }
1844
1845     /**
1846      * Checks if the instant of this date-time is equal to that of the specified date-time.

```

```

1847 * <p>
1848 * This method differs from the comparison in {@link #compareTo} and {@link #equals}
1849 * in that it only compares the instant of the date-time. This is equivalent to using
1850 * {@code dateTime1.toInstant().equals(dateTime2.toInstant());}.
1851 *
1852 * @param other the other date-time to compare to, not null
1853 * @return true if the instant equals the instant of the specified date-time
1854 */
1855 public boolean isEqual(OffsetDateTime other) {
1856     return toEpochSecond() == other.toEpochSecond() &&
1857         toLocalTime().getNano() == other.toLocalTime().getNano();
1858 }
1859
1860 //-----
1861 /**
1862 * Checks if this date-time is equal to another date-time.
1863 * <p>
1864 * The comparison is based on the local date-time and the offset.
1865 * To compare for the same instant on the time-line, use {@link #isEqual}.
1866 * Only objects of type {@code OffsetDateTime} are compared, other types return false.
1867 *
1868 * @param obj the object to check, null returns false
1869 * @return true if this is equal to the other date-time
1870 */
1871 @Override
1872 public boolean equals(Object obj) {
1873     if (this == obj) {
1874         return true;
1875     }
1876     if (obj instanceof OffsetDateTime) {
1877         OffsetDateTime other = (OffsetDateTime) obj;
1878         return dateTime.equals(other.dateTime) && offset.equals(other.offset);
1879     }
1880     return false;
1881 }
1882
1883 /**
1884 * A hash code for this date-time.
1885 *
1886 * @return a suitable hash code
1887 */
1888 @Override
1889 public int hashCode() {
1890     return dateTime.hashCode() ^ offset.hashCode();
1891 }
1892
1893 //-----
1894 /**
1895 * Outputs this date-time as a {@code String}, such as {@code 2007-12-03T10:15:30+01:00}.
1896 * <p>
1897 * The output will be one of the following ISO-8601 formats:
1898 * <ul>
1899 * <li>{@code uuuu-MM-dd'T'HH:mmXXXXX}</li>

```

```

1900 * <li>{@code uuuu-MM-dd'T'HH:mm:ssXXXXX}</li>
1901 * <li>{@code uuuu-MM-dd'T'HH:mm:ss.SSSXXXXX}</li>
1902 * <li>{@code uuuu-MM-dd'T'HH:mm:ss.SSSSSXXXXX}</li>
1903 * <li>{@code uuuu-MM-dd'T'HH:mm:ss.SSSSSSSXXXXX}</li>
1904 * </ul>
1905 * The format used will be the shortest that outputs the full value of
1906 * the time where the omitted parts are implied to be zero.
1907 *
1908 * @return a string representation of this date-time, not null
1909 */
1910 @Override
1911 public String toString() {
1912     return dateTime.toString() + offset.toString();
1913 }
1914
1915 //-----
1916 /**
1917  * Writes the object using a
1918  * <a href=".../serialized-form.html#java.time.Ser">dedicated serialized form</a>.
1919  * @serialData
1920  * <pre>
1921  *   out.writeByte(10); // identifies an OffsetDateTime
1922  *   // the <a href=".../serialized-form.html#java.time.LocalDateTime">datetime</a> excluding
1923  *       the one byte header
1924  *   // the <a href=".../serialized-form.html#java.time.ZoneOffset">offset</a> excluding the
1925  *       one byte header
1926  * </pre>
1927  *
1928  * @return the instance of {@code Ser}, not null
1929  */
1930 private Object writeReplace() {
1931     return new Ser(Ser.OFFSET_DATE_TIME_TYPE, this);
1932 }
1933
1934 /**
1935  * Defend against malicious streams.
1936  *
1937  * @param s the stream to read
1938  * @throws InvalidObjectException always
1939  */
1940 private void readObject(ObjectInputStream s) throws InvalidObjectException {
1941     throw new InvalidObjectException("Deserialization via serialization delegate");
1942 }
1943
1944 void writeExternal(ObjectOutput out) throws IOException {
1945     dateTime.writeExternal(out);
1946     offset.writeExternal(out);
1947 }
1948
1949 static OffsetDateTime readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
1950     LocalDateTime dateTime = LocalDateTime.readExternal(in);
1951     ZoneOffset offset = ZoneOffset.readExternal(in);
1952     return OffsetDateTime.of(dateTime, offset);

```

```
1951     }  
1952  
1953 }
```

4.12 OffsetTime.java

Secuencia 61: java/time/OffsetTime.java

```
1  /*  
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.  
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.  
4  *  
5  *  
6  *  
7  *  
8  *  
9  *  
10 *  
11 *  
12 *  
13 *  
14 *  
15 *  
16 *  
17 *  
18 *  
19 *  
20 *  
21 *  
22 *  
23 *  
24 */  
25  
26 /*  
27 *  
28 *  
29 *  
30 *  
31 *  
32 * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos  
33 *  
34 * All rights reserved.  
35 *  
36 * Redistribution and use in source and binary forms, with or without  
37 * modification, are permitted provided that the following conditions are met:  
38 *  
39 * * Redistributions of source code must retain the above copyright notice,  
40 *   this list of conditions and the following disclaimer.  
41 *  
42 * * Redistributions in binary form must reproduce the above copyright notice,  
43 *   this list of conditions and the following disclaimer in the documentation  
44 *   and/or other materials provided with the distribution.  
45 *  
46 * * Neither the name of JSR-310 nor the names of its contributors
```



```
47 *    may be used to endorse or promote products derived from this software
48 *    without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
62 package java.time;
63
64 import static java.time.LocalDateTime.NANOS_PER_HOUR;
65 import static java.time.LocalDateTime.NANOS_PER_MINUTE;
66 import static java.time.LocalDateTime.NANOS_PER_SECOND;
67 import static java.time.LocalDateTime.SECONDS_PER_DAY;
68 import static java.time.temporal.ChronoField.NANO_OF_DAY;
69 import static java.time.temporal.ChronoField.OFFSET_SECONDS;
70 import static java.time.temporal.ChronoUnit.NANOS;
71
72 import java.io.IOException;
73 import java.io.ObjectInput;
74 import java.io.ObjectOutput;
75 import java.io.InvalidObjectException;
76 import java.io.ObjectInputStream;
77 import java.io.Serializable;
78 import java.time.format.DateTimeFormatter;
79 import java.time.format.DateTimeParseException;
80 import java.time.temporal.ChronoField;
81 import java.time.temporal.ChronoUnit;
82 import java.time.temporal.Temporal;
83 import java.time.temporal.TemporalAccessor;
84 import java.time.temporal.TemporalAdjuster;
85 import java.time.temporal.TemporalAmount;
86 import java.time.temporal.TemporalField;
87 import java.time.temporal.TemporalQueries;
88 import java.time.temporal.TemporalQuery;
89 import java.time.temporal.TemporalUnit;
90 import java.time.temporal.UnsupportedTemporalTypeException;
91 import java.time.temporal.ValueRange;
92 import java.time.zone.ZoneRules;
93 import java.util.Objects;
94
95 /**
96  * A time with an offset from UTC/Greenwich in the ISO-8601 calendar system,
97  * such as {@code 10:15:30+01:00}.
98  * <p>
99  * {@code OffsetTime} is an immutable date-time object that represents a time, often
```

```

100 * viewed as hour-minute-second-offset.
101 * This class stores all time fields, to a precision of nanoseconds,
102 * as well as a zone offset.
103 * For example, the value "13:45.30.123456789+02:00" can be stored
104 * in an {@code OffsetTime}.
105 *
106 * <p>
107 * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
108 * class; use of identity-sensitive operations (including reference equality
109 * ({@code ==}), identity hash code, or synchronization) on instances of
110 * {@code OffsetTime} may have unpredictable results and should be avoided.
111 * The {@code equals} method should be used for comparisons.
112 *
113 * @implSpec
114 * This class is immutable and thread-safe.
115 *
116 * @since 1.8
117 */
118 public final class OffsetTime
119     implements Temporal, TemporalAdjuster, Comparable<OffsetTime>, Serializable {
120
121     /**
122      * The minimum supported {@code OffsetTime}, '00:00:00+18:00'.
123      * This is the time of midnight at the start of the day in the maximum offset
124      * (larger offsets are earlier on the time-line).
125      * This combines {@link LocalDateTime#MIN} and {@link ZoneOffset#MAX}.
126      * This could be used by an application as a "far past" date.
127      */
128     public static final OffsetTime MIN = LocalDateTime.MIN.atOffset(ZoneOffset.MAX);
129
130     /**
131      * The maximum supported {@code OffsetTime}, '23:59:59.999999999-18:00'.
132      * This is the time just before midnight at the end of the day in the minimum offset
133      * (larger negative offsets are later on the time-line).
134      * This combines {@link LocalDateTime#MAX} and {@link ZoneOffset#MIN}.
135      * This could be used by an application as a "far future" date.
136      */
137     public static final OffsetTime MAX = LocalDateTime.MAX.atOffset(ZoneOffset.MIN);
138
139     /**
140      * Serialization version.
141      */
142     private static final long serialVersionUID = 7264499704384272492L;
143
144     /**
145      * The local date-time.
146      */
147     private final LocalDateTime time;
148
149     /**
150      * The offset from UTC/Greenwich.
151      */
152     private final ZoneOffset offset;
153
154     //-----

```

```

153  /**
154      * Obtains the current time from the system clock in the default time-zone.
155      * <p>
156      * This will query the {@link Clock#systemDefaultZone() system clock} in the default
157      * time-zone to obtain the current time.
158      * The offset will be calculated from the time-zone in the clock.
159      * <p>
160      * Using this method will prevent the ability to use an alternate clock for testing
161      * because the clock is hard-coded.
162      *
163      * @return the current time using the system clock and default time-zone, not null
164      */
165  public static OffsetTime now() {
166      return now(Clock.systemDefaultZone());
167  }
168
169  /**
170      * Obtains the current time from the system clock in the specified time-zone.
171      * <p>
172      * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current time.
173      * Specifying the time-zone avoids dependence on the default time-zone.
174      * The offset will be calculated from the specified time-zone.
175      * <p>
176      * Using this method will prevent the ability to use an alternate clock for testing
177      * because the clock is hard-coded.
178      *
179      * @param zone the zone ID to use, not null
180      * @return the current time using the system clock, not null
181      */
182  public static OffsetTime now(ZoneId zone) {
183      return now(Clock.system(zone));
184  }
185
186  /**
187      * Obtains the current time from the specified clock.
188      * <p>
189      * This will query the specified clock to obtain the current time.
190      * The offset will be calculated from the time-zone in the clock.
191      * <p>
192      * Using this method allows the use of an alternate clock for testing.
193      * The alternate clock may be introduced using {@link Clock dependency injection}.
194      *
195      * @param clock the clock to use, not null
196      * @return the current time, not null
197      */
198  public static OffsetTime now(Clock clock) {
199      Objects.requireNonNull(clock, "clock");
200      final Instant now = clock.instant(); // called once
201      return ofInstant(now, clock.getZone().getRules().getOffset(now));
202  }
203
204  //-----
205  /**

```

```

206     * Obtains an instance of {@code OffsetTime} from a local time and an offset.
207     *
208     * @param time    the local time, not null
209     * @param offset  the zone offset, not null
210     * @return the offset time, not null
211     */
212     public static OffsetTime of(LocalTime time, ZoneOffset offset) {
213         return new OffsetTime(time, offset);
214     }
215
216     /**
217     * Obtains an instance of {@code OffsetTime} from an hour, minute, second and nanosecond.
218     * <p>
219     * This creates an offset time with the four specified fields.
220     * <p>
221     * This method exists primarily for writing test cases.
222     * Non test-code will typically use other methods to create an offset time.
223     * {@code LocalTime} has two additional convenience variants of the
224     * equivalent factory method taking fewer arguments.
225     * They are not provided here to reduce the footprint of the API.
226     *
227     * @param hour    the hour-of-day to represent, from 0 to 23
228     * @param minute  the minute-of-hour to represent, from 0 to 59
229     * @param second  the second-of-minute to represent, from 0 to 59
230     * @param nanoOfSecond  the nano-of-second to represent, from 0 to 999,999,999
231     * @param offset  the zone offset, not null
232     * @return the offset time, not null
233     * @throws DateTimeException if the value of any field is out of range
234     */
235     public static OffsetTime of(int hour, int minute, int second, int nanoOfSecond, ZoneOffset
        offset) {
236         return new OffsetTime(LocalTime.of(hour, minute, second, nanoOfSecond), offset);
237     }
238
239     //-----
240     /**
241     * Obtains an instance of {@code OffsetTime} from an {@code Instant} and zone ID.
242     * <p>
243     * This creates an offset time with the same instant as that specified.
244     * Finding the offset from UTC/Greenwich is simple as there is only one valid
245     * offset for each instant.
246     * <p>
247     * The date component of the instant is dropped during the conversion.
248     * This means that the conversion can never fail due to the instant being
249     * out of the valid range of dates.
250     *
251     * @param instant  the instant to create the time from, not null
252     * @param zone     the time-zone, which may be an offset, not null
253     * @return the offset time, not null
254     */
255     public static OffsetTime ofInstant(Instant instant, ZoneId zone) {
256         Objects.requireNonNull(instant, "instant");
257         Objects.requireNonNull(zone, "zone");

```

```

258     ZoneRules rules = zone.getRules();
259     ZoneOffset offset = rules.getOffset(instant);
260     long localSecond = instant.getEpochSecond() + offset.getTotalSeconds(); // overflow caught
        later
261     int secsOfDay = (int) Math.floorMod(localSecond, SECONDS_PER_DAY);
262     LocalTime time = LocalTime.ofNanoOfDay(secsOfDay * NANOS_PER_SECOND + instant.getNano());
263     return new OffsetTime(time, offset);
264 }
265
266 //-----
267 /**
268  * Obtains an instance of {@code OffsetTime} from a temporal object.
269  * <p>
270  * This obtains an offset time based on the specified temporal.
271  * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
272  * which this factory converts to an instance of {@code OffsetTime}.
273  * <p>
274  * The conversion extracts and combines the {@code ZoneOffset} and the
275  * {@code LocalTime} from the temporal object.
276  * Implementations are permitted to perform optimizations such as accessing
277  * those fields that are equivalent to the relevant objects.
278  * <p>
279  * This method matches the signature of the functional interface {@link TemporalQuery}
280  * allowing it to be used as a query via method reference, {@code OffsetTime::from}.
281  *
282  * @param temporal the temporal object to convert, not null
283  * @return the offset time, not null
284  * @throws DateTimeException if unable to convert to an {@code OffsetTime}
285  */
286 public static OffsetTime from(TemporalAccessor temporal) {
287     if (temporal instanceof OffsetTime) {
288         return (OffsetTime) temporal;
289     }
290     try {
291         LocalTime time = LocalTime.from(temporal);
292         ZoneOffset offset = ZoneOffset.from(temporal);
293         return new OffsetTime(time, offset);
294     } catch (DateTimeException ex) {
295         throw new DateTimeException("Unable to obtain OffsetTime from TemporalAccessor: " +
296             temporal + " of type " + temporal.getClass().getName(), ex);
297     }
298 }
299
300 //-----
301 /**
302  * Obtains an instance of {@code OffsetTime} from a text string such as {@code 10:15:30+01:00}.
303  * <p>
304  * The string must represent a valid time and is parsed using
305  * {@link java.time.format.DateTimeFormatter#ISO_OFFSET_TIME}.
306  *
307  * @param text the text to parse such as "10:15:30+01:00", not null
308  * @return the parsed local time, not null
309  * @throws DateTimeParseException if the text cannot be parsed

```

```

310     */
311     public static OffsetTime parse(CharSequence text) {
312         return parse(text, DateTimeFormatter.ISO_OFFSET_TIME);
313     }
314
315     /**
316      * Obtains an instance of {@code OffsetTime} from a text string using a specific formatter.
317      * <p>
318      * The text is parsed using the formatter, returning a time.
319      *
320      * @param text the text to parse, not null
321      * @param formatter the formatter to use, not null
322      * @return the parsed offset time, not null
323      * @throws DateTimeParseException if the text cannot be parsed
324      */
325     public static OffsetTime parse(CharSequence text, DateTimeFormatter formatter) {
326         Objects.requireNonNull(formatter, "formatter");
327         return formatter.parse(text, OffsetTime::from);
328     }
329
330     //-----
331     /**
332      * Constructor.
333      *
334      * @param time the local time, not null
335      * @param offset the zone offset, not null
336      */
337     private OffsetTime(LocalTime time, ZoneOffset offset) {
338         this.time = Objects.requireNonNull(time, "time");
339         this.offset = Objects.requireNonNull(offset, "offset");
340     }
341
342     /**
343      * Returns a new time based on this one, returning {@code this} where possible.
344      *
345      * @param time the time to create with, not null
346      * @param offset the zone offset to create with, not null
347      */
348     private OffsetTime with(LocalTime time, ZoneOffset offset) {
349         if (this.time == time && this.offset.equals(offset)) {
350             return this;
351         }
352         return new OffsetTime(time, offset);
353     }
354
355     //-----
356     /**
357      * Checks if the specified field is supported.
358      * <p>
359      * This checks if this time can be queried for the specified field.
360      * If false, then calling the {@link #range(TemporalField) range},
361      * {@link #get(TemporalField) get} and {@link #with(TemporalField, long)}
362      * methods will throw an exception.

```

```

363 * <p>
364 * If the field is a {@link ChronoField} then the query is implemented here.
365 * The supported fields are:
366 * <ul>
367 * <li>{@code NANO_OF_SECOND}
368 * <li>{@code NANO_OF_DAY}
369 * <li>{@code MICRO_OF_SECOND}
370 * <li>{@code MICRO_OF_DAY}
371 * <li>{@code MILLI_OF_SECOND}
372 * <li>{@code MILLI_OF_DAY}
373 * <li>{@code SECOND_OF_MINUTE}
374 * <li>{@code SECOND_OF_DAY}
375 * <li>{@code MINUTE_OF_HOUR}
376 * <li>{@code MINUTE_OF_DAY}
377 * <li>{@code HOUR_OF_AMPM}
378 * <li>{@code CLOCK_HOUR_OF_AMPM}
379 * <li>{@code HOUR_OF_DAY}
380 * <li>{@code CLOCK_HOUR_OF_DAY}
381 * <li>{@code AMPM_OF_DAY}
382 * <li>{@code OFFSET_SECONDS}
383 * </ul>
384 * All other {@code ChronoField} instances will return false.
385 * <p>
386 * If the field is not a {@code ChronoField}, then the result of this method
387 * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
388 * passing {@code this} as the argument.
389 * Whether the field is supported is determined by the field.
390 *
391 * @param field the field to check, null returns false
392 * @return true if the field is supported on this time, false if not
393 */
394 @Override
395 public boolean isSupported(TemporalField field) {
396     if (field instanceof ChronoField) {
397         return field.isTimeBased() || field == OFFSET_SECONDS;
398     }
399     return field != null && field.isSupportedBy(this);
400 }
401
402 /**
403 * Checks if the specified unit is supported.
404 * <p>
405 * This checks if the specified unit can be added to, or subtracted from, this offset-time.
406 * If false, then calling the {@link #plus(long, TemporalUnit)} and
407 * {@link #minus(long, TemporalUnit) minus} methods will throw an exception.
408 * <p>
409 * If the unit is a {@link ChronoUnit} then the query is implemented here.
410 * The supported units are:
411 * <ul>
412 * <li>{@code NANOS}
413 * <li>{@code MICROS}
414 * <li>{@code MILLIS}
415 * <li>{@code SECONDS}

```

```

416 * <li>{@code MINUTES}
417 * <li>{@code HOURS}
418 * <li>{@code HALF_DAYS}
419 * </ul>
420 * All other {@code ChronoUnit} instances will return false.
421 * <p>
422 * If the unit is not a {@code ChronoUnit}, then the result of this method
423 * is obtained by invoking {@code TemporalUnit.isSupportedBy(Temporal)}
424 * passing {@code this} as the argument.
425 * Whether the unit is supported is determined by the unit.
426 *
427 * @param unit the unit to check, null returns false
428 * @return true if the unit can be added/subtracted, false if not
429 */
430 @Override // override for Javadoc
431 public boolean isSupported(TemporalUnit unit) {
432     if (unit instanceof ChronoUnit) {
433         return unit.isTimeBased();
434     }
435     return unit != null && unit.isSupportedBy(this);
436 }
437
438 //-----
439 /**
440 * Gets the range of valid values for the specified field.
441 * <p>
442 * The range object expresses the minimum and maximum valid values for a field.
443 * This time is used to enhance the accuracy of the returned range.
444 * If it is not possible to return the range, because the field is not supported
445 * or for some other reason, an exception is thrown.
446 * <p>
447 * If the field is a {@link ChronoField} then the query is implemented here.
448 * The {@link #isSupported(TemporalField) supported fields} will return
449 * appropriate range instances.
450 * All other {@code ChronoField} instances will throw an {@code
451     UnsupportedOperationException}.
452 * <p>
453 * If the field is not a {@code ChronoField}, then the result of this method
454 * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
455 * passing {@code this} as the argument.
456 * Whether the range can be obtained is determined by the field.
457 *
458 * @param field the field to query the range for, not null
459 * @return the range of valid values for the field, not null
460 * @throws DateTimeException if the range for the field cannot be obtained
461 * @throws UnsupportedOperationException if the field is not supported
462 */
463 @Override
464 public ValueRange range(TemporalField field) {
465     if (field instanceof ChronoField) {
466         if (field == OFFSET_SECONDS) {
467             return field.range();
468         }
469     }
470 }

```



```

468         return time.range(field);
469     }
470     return field.rangeRefinedBy(this);
471 }
472
473 /**
474  * Gets the value of the specified field from this time as an {@code int}.
475  * <p>
476  * This queries this time for the value of the specified field.
477  * The returned value will always be within the valid range of values for the field.
478  * If it is not possible to return the value, because the field is not supported
479  * or for some other reason, an exception is thrown.
480  * <p>
481  * If the field is a {@link ChronoField} then the query is implemented here.
482  * The {@link #isSupported(TemporalField) supported fields} will return valid
483  * values based on this time, except {@code NANO_OF_DAY} and {@code MICRO_OF_DAY}
484  * which are too large to fit in an {@code int} and throw a {@code DateTimeException}.
485  * All other {@code ChronoField} instances will throw an {@code
486     * UnsupportedTemporalTypeException}.
487  * <p>
488  * If the field is not a {@code ChronoField}, then the result of this method
489  * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
490  * passing {@code this} as the argument. Whether the value can be obtained,
491  * and what the value represents, is determined by the field.
492  *
493  * @param field the field to get, not null
494  * @return the value for the field
495  * @throws DateTimeException if a value for the field cannot be obtained or
496  *         the value is outside the range of valid values for the field
497  * @throws UnsupportedTemporalTypeException if the field is not supported or
498  *         the range of values exceeds an {@code int}
499  * @throws ArithmeticException if numeric overflow occurs
500  */
501 @Override // override for Javadoc
502 public int get(TemporalField field) {
503     return Temporal.super.get(field);
504 }
505
506 /**
507  * Gets the value of the specified field from this time as a {@code long}.
508  * <p>
509  * This queries this time for the value of the specified field.
510  * If it is not possible to return the value, because the field is not supported
511  * or for some other reason, an exception is thrown.
512  * <p>
513  * If the field is a {@link ChronoField} then the query is implemented here.
514  * The {@link #isSupported(TemporalField) supported fields} will return valid
515  * values based on this time.
516  * All other {@code ChronoField} instances will throw an {@code
517     * UnsupportedTemporalTypeException}.
518  * <p>
519  * If the field is not a {@code ChronoField}, then the result of this method
520  * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}

```

```

519     * passing {@code this} as the argument. Whether the value can be obtained,
520     * and what the value represents, is determined by the field.
521     *
522     * @param field the field to get, not null
523     * @return the value for the field
524     * @throws DateTimeException if a value for the field cannot be obtained
525     * @throws UnsupportedTemporalTypeException if the field is not supported
526     * @throws ArithmeticException if numeric overflow occurs
527     */
528     @Override
529     public long getLong(TemporalField field) {
530         if (field instanceof ChronoField) {
531             if (field == OFFSET_SECONDS) {
532                 return offset.getTotalSeconds();
533             }
534             return time.getLong(field);
535         }
536         return field.getFrom(this);
537     }
538
539     //-----
540     /**
541     * Gets the zone offset, such as '+01:00'.
542     * <p>
543     * This is the offset of the local time from UTC/Greenwich.
544     *
545     * @return the zone offset, not null
546     */
547     public ZoneOffset getOffset() {
548         return offset;
549     }
550
551     /**
552     * Returns a copy of this {@code OffsetTime} with the specified offset ensuring
553     * that the result has the same local time.
554     * <p>
555     * This method returns an object with the same {@code LocalTime} and the specified {@code
556     * ZoneOffset}.
557     * No calculation is needed or performed.
558     * For example, if this time represents {@code 10:30+02:00} and the offset specified is
559     * {@code +03:00}, then this method will return {@code 10:30+03:00}.
560     * <p>
561     * To take into account the difference between the offsets, and adjust the time fields,
562     * use {@link #withOffsetSameInstant}.
563     * <p>
564     * This instance is immutable and unaffected by this method call.
565     *
566     * @param offset the zone offset to change to, not null
567     * @return an {@code OffsetTime} based on this time with the requested offset, not null
568     */
569     public OffsetTime withOffsetSameLocal(ZoneOffset offset) {
570         return offset != null && offset.equals(this.offset) ? this : new OffsetTime(time, offset);
571     }

```

```

571
572 /**
573  * Returns a copy of this {@code OffsetTime} with the specified offset ensuring
574  * that the result is at the same instant on an implied day.
575  * <p>
576  * This method returns an object with the specified {@code ZoneOffset} and a {@code LocalTime}
577  * adjusted by the difference between the two offsets.
578  * This will result in the old and new objects representing the same instant on an implied day.
579  * This is useful for finding the local time in a different offset.
580  * For example, if this time represents {@code 10:30+02:00} and the offset specified is
581  * {@code +03:00}, then this method will return {@code 11:30+03:00}.
582  * <p>
583  * To change the offset without adjusting the local time use {@link #withOffsetSameLocal}.
584  * <p>
585  * This instance is immutable and unaffected by this method call.
586  *
587  * @param offset the zone offset to change to, not null
588  * @return an {@code OffsetTime} based on this time with the requested offset, not null
589  */
590 public OffsetTime withOffsetSameInstant(ZoneOffset offset) {
591     if (offset.equals(this.offset)) {
592         return this;
593     }
594     int difference = offset.getTotalSeconds() - this.offset.getTotalSeconds();
595     LocalTime adjusted = time.plusSeconds(difference);
596     return new OffsetTime(adjusted, offset);
597 }
598
599 //-----
600 /**
601  * Gets the {@code LocalTime} part of this date-time.
602  * <p>
603  * This returns a {@code LocalTime} with the same hour, minute, second and
604  * nanosecond as this date-time.
605  *
606  * @return the time part of this date-time, not null
607  */
608 public LocalTime toLocalTime() {
609     return time;
610 }
611
612 //-----
613 /**
614  * Gets the hour-of-day field.
615  *
616  * @return the hour-of-day, from 0 to 23
617  */
618 public int getHour() {
619     return time.getHour();
620 }
621
622 /**
623  * Gets the minute-of-hour field.

```

```

624     *
625     * @return the minute-of-hour, from 0 to 59
626     */
627     public int getMinute() {
628         return time.getMinute();
629     }
630
631     /**
632     * Gets the second-of-minute field.
633     *
634     * @return the second-of-minute, from 0 to 59
635     */
636     public int getSecond() {
637         return time.getSecond();
638     }
639
640     /**
641     * Gets the nano-of-second field.
642     *
643     * @return the nano-of-second, from 0 to 999,999,999
644     */
645     public int getNano() {
646         return time.getNano();
647     }
648
649     //-----
650     /**
651     * Returns an adjusted copy of this time.
652     * <p>
653     * This returns an {@code OffsetTime}, based on this one, with the time adjusted.
654     * The adjustment takes place using the specified adjuster strategy object.
655     * Read the documentation of the adjuster to understand what adjustment will be made.
656     * <p>
657     * A simple adjuster might simply set the one of the fields, such as the hour field.
658     * A more complex adjuster might set the time to the last hour of the day.
659     * <p>
660     * The classes {@link LocalDateTime} and {@link ZoneOffset} implement {@code TemporalAdjuster},
661     * thus this method can be used to change the time or offset:
662     * <pre>
663     *     result = offsetTime.with(time);
664     *     result = offsetTime.with(offset);
665     * </pre>
666     * <p>
667     * The result of this method is obtained by invoking the
668     * {@link TemporalAdjuster#adjustInto(Temporal)} method on the
669     * specified adjuster passing {@code this} as the argument.
670     * <p>
671     * This instance is immutable and unaffected by this method call.
672     *
673     * @param adjuster the adjuster to use, not null
674     * @return an {@code OffsetTime} based on {@code this} with the adjustment made, not null
675     * @throws DateTimeException if the adjustment cannot be made
676     * @throws ArithmeticException if numeric overflow occurs

```

```

677     */
678     @Override
679     public OffsetTime with(TemporalAdjuster adjuster) {
680         // optimizations
681         if (adjuster instanceof LocalTime) {
682             return with((LocalTime) adjuster, offset);
683         } else if (adjuster instanceof ZoneOffset) {
684             return with(time, (ZoneOffset) adjuster);
685         } else if (adjuster instanceof OffsetTime) {
686             return (OffsetTime) adjuster;
687         }
688         return (OffsetTime) adjuster.adjustInto(this);
689     }
690
691     /**
692      * Returns a copy of this time with the specified field set to a new value.
693      * <p>
694      * This returns an {@code OffsetTime}, based on this one, with the value
695      * for the specified field changed.
696      * This can be used to change any supported field, such as the hour, minute or second.
697      * If it is not possible to set the value, because the field is not supported or for
698      * some other reason, an exception is thrown.
699      * <p>
700      * If the field is a {@link ChronoField} then the adjustment is implemented here.
701      * <p>
702      * The {@code OFFSET_SECONDS} field will return a time with the specified offset.
703      * The local time is unaltered. If the new offset value is outside the valid range
704      * then a {@code DateTimeException} will be thrown.
705      * <p>
706      * The other {@link #isSupported(TemporalField) supported fields} will behave as per
707      * the matching method on {@link LocalTime#with(TemporalField, long)} LocalTime}.
708      * In this case, the offset is not part of the calculation and will be unchanged.
709      * <p>
710      * All other {@code ChronoField} instances will throw an {@code
711          UnsupportedTemporalTypeException}.
712      * <p>
713      * If the field is not a {@code ChronoField}, then the result of this method
714      * is obtained by invoking {@code TemporalField.adjustInto(Temporal, long)}
715      * passing {@code this} as the argument. In this case, the field determines
716      * whether and how to adjust the instant.
717      * <p>
718      * This instance is immutable and unaffected by this method call.
719      *
720      * @param field the field to set in the result, not null
721      * @param newValue the new value of the field in the result
722      * @return an {@code OffsetTime} based on {@code this} with the specified field set, not null
723      * @throws DateTimeException if the field cannot be set
724      * @throws UnsupportedTemporalTypeException if the field is not supported
725      * @throws ArithmeticException if numeric overflow occurs
726      */
727     @Override
728     public OffsetTime with(TemporalField field, long newValue) {
729         if (field instanceof ChronoField) {

```

```

729         if (field == OFFSET_SECONDS) {
730             ChronoField f = (ChronoField) field;
731             return with(time, ZoneOffset.ofTotalSeconds(f.checkValidIntValue(newValue)));
732         }
733         return with(time.with(field, newValue), offset);
734     }
735     return field.adjustInto(this, newValue);
736 }
737
738 //-----
739 /**
740  * Returns a copy of this {@code OffsetTime} with the hour-of-day altered.
741  * <p>
742  * The offset does not affect the calculation and will be the same in the result.
743  * <p>
744  * This instance is immutable and unaffected by this method call.
745  *
746  * @param hour the hour-of-day to set in the result, from 0 to 23
747  * @return an {@code OffsetTime} based on this time with the requested hour, not null
748  * @throws DateTimeException if the hour value is invalid
749  */
750 public OffsetTime withHour(int hour) {
751     return with(time.withHour(hour), offset);
752 }
753
754 /**
755  * Returns a copy of this {@code OffsetTime} with the minute-of-hour altered.
756  * <p>
757  * The offset does not affect the calculation and will be the same in the result.
758  * <p>
759  * This instance is immutable and unaffected by this method call.
760  *
761  * @param minute the minute-of-hour to set in the result, from 0 to 59
762  * @return an {@code OffsetTime} based on this time with the requested minute, not null
763  * @throws DateTimeException if the minute value is invalid
764  */
765 public OffsetTime withMinute(int minute) {
766     return with(time.withMinute(minute), offset);
767 }
768
769 /**
770  * Returns a copy of this {@code OffsetTime} with the second-of-minute altered.
771  * <p>
772  * The offset does not affect the calculation and will be the same in the result.
773  * <p>
774  * This instance is immutable and unaffected by this method call.
775  *
776  * @param second the second-of-minute to set in the result, from 0 to 59
777  * @return an {@code OffsetTime} based on this time with the requested second, not null
778  * @throws DateTimeException if the second value is invalid
779  */
780 public OffsetTime withSecond(int second) {
781     return with(time.withSecond(second), offset);

```

```

782     }
783
784     /**
785      * Returns a copy of this {@code OffsetTime} with the nano-of-second altered.
786      * <p>
787      * The offset does not affect the calculation and will be the same in the result.
788      * <p>
789      * This instance is immutable and unaffected by this method call.
790      *
791      * @param nanoOfSecond the nano-of-second to set in the result, from 0 to 999,999,999
792      * @return an {@code OffsetTime} based on this time with the requested nanosecond, not null
793      * @throws DateTimeException if the nanos value is invalid
794      */
795     public OffsetTime withNano(int nanoOfSecond) {
796         return with(time.withNano(nanoOfSecond), offset);
797     }
798
799     //-----
800     /**
801      * Returns a copy of this {@code OffsetTime} with the time truncated.
802      * <p>
803      * Truncation returns a copy of the original time with fields
804      * smaller than the specified unit set to zero.
805      * For example, truncating with the {@link ChronoUnit#MINUTES minutes} unit
806      * will set the second-of-minute and nano-of-second field to zero.
807      * <p>
808      * The unit must have a {@linkplain TemporalUnit#getDuration() duration}
809      * that divides into the length of a standard day without remainder.
810      * This includes all supplied time units on {@link ChronoUnit} and
811      * {@link ChronoUnit#DAYS DAYS}. Other units throw an exception.
812      * <p>
813      * The offset does not affect the calculation and will be the same in the result.
814      * <p>
815      * This instance is immutable and unaffected by this method call.
816      *
817      * @param unit the unit to truncate to, not null
818      * @return an {@code OffsetTime} based on this time with the time truncated, not null
819      * @throws DateTimeException if unable to truncate
820      * @throws UnsupportedTemporalTypeException if the unit is not supported
821      */
822     public OffsetTime truncatedTo(TemporalUnit unit) {
823         return with(time.truncatedTo(unit), offset);
824     }
825
826     //-----
827     /**
828      * Returns a copy of this time with the specified amount added.
829      * <p>
830      * This returns an {@code OffsetTime}, based on this one, with the specified amount added.
831      * The amount is typically {@link Duration} but may be any other type implementing
832      * the {@link TemporalAmount} interface.
833      * <p>
834      * The calculation is delegated to the amount object by calling

```

```

835 * {@link TemporalAmount#addTo(Temporal)}. The amount implementation is free
836 * to implement the addition in any way it wishes, however it typically
837 * calls back to {@link #plus(long, TemporalUnit)}. Consult the documentation
838 * of the amount implementation to determine if it can be successfully added.
839 * <p>
840 * This instance is immutable and unaffected by this method call.
841 *
842 * @param amountToAdd the amount to add, not null
843 * @return an {@code OffsetTime} based on this time with the addition made, not null
844 * @throws DateTimeException if the addition cannot be made
845 * @throws ArithmeticException if numeric overflow occurs
846 */
847 @Override
848 public OffsetTime plus(TemporalAmount amountToAdd) {
849     return (OffsetTime) amountToAdd.addTo(this);
850 }
851
852 /**
853 * Returns a copy of this time with the specified amount added.
854 * <p>
855 * This returns an {@code OffsetTime}, based on this one, with the amount
856 * in terms of the unit added. If it is not possible to add the amount, because the
857 * unit is not supported or for some other reason, an exception is thrown.
858 * <p>
859 * If the field is a {@link ChronoUnit} then the addition is implemented by
860 * {@link LocalDateTime#plus(long, TemporalUnit)}.
861 * The offset is not part of the calculation and will be unchanged in the result.
862 * <p>
863 * If the field is not a {@code ChronoUnit}, then the result of this method
864 * is obtained by invoking {@code TemporalUnit.addTo(Temporal, long)}
865 * passing {@code this} as the argument. In this case, the unit determines
866 * whether and how to perform the addition.
867 * <p>
868 * This instance is immutable and unaffected by this method call.
869 *
870 * @param amountToAdd the amount of the unit to add to the result, may be negative
871 * @param unit the unit of the amount to add, not null
872 * @return an {@code OffsetTime} based on this time with the specified amount added, not null
873 * @throws DateTimeException if the addition cannot be made
874 * @throws UnsupportedTemporalTypeException if the unit is not supported
875 * @throws ArithmeticException if numeric overflow occurs
876 */
877 @Override
878 public OffsetTime plus(long amountToAdd, TemporalUnit unit) {
879     if (unit instanceof ChronoUnit) {
880         return with(time.plus(amountToAdd, unit), offset);
881     }
882     return unit.addTo(this, amountToAdd);
883 }
884
885 //-----
886 /**
887 * Returns a copy of this {@code OffsetTime} with the specified number of hours added.

```



```
888 * <p>
889 * This adds the specified number of hours to this time, returning a new time.
890 * The calculation wraps around midnight.
891 * <p>
892 * This instance is immutable and unaffected by this method call.
893 *
894 * @param hours the hours to add, may be negative
895 * @return an {@code OffsetTime} based on this time with the hours added, not null
896 */
897 public OffsetTime plusHours(long hours) {
898     return with(time.plusHours(hours), offset);
899 }
900
901 /**
902 * Returns a copy of this {@code OffsetTime} with the specified number of minutes added.
903 * <p>
904 * This adds the specified number of minutes to this time, returning a new time.
905 * The calculation wraps around midnight.
906 * <p>
907 * This instance is immutable and unaffected by this method call.
908 *
909 * @param minutes the minutes to add, may be negative
910 * @return an {@code OffsetTime} based on this time with the minutes added, not null
911 */
912 public OffsetTime plusMinutes(long minutes) {
913     return with(time.plusMinutes(minutes), offset);
914 }
915
916 /**
917 * Returns a copy of this {@code OffsetTime} with the specified number of seconds added.
918 * <p>
919 * This adds the specified number of seconds to this time, returning a new time.
920 * The calculation wraps around midnight.
921 * <p>
922 * This instance is immutable and unaffected by this method call.
923 *
924 * @param seconds the seconds to add, may be negative
925 * @return an {@code OffsetTime} based on this time with the seconds added, not null
926 */
927 public OffsetTime plusSeconds(long seconds) {
928     return with(time.plusSeconds(seconds), offset);
929 }
930
931 /**
932 * Returns a copy of this {@code OffsetTime} with the specified number of nanoseconds added.
933 * <p>
934 * This adds the specified number of nanoseconds to this time, returning a new time.
935 * The calculation wraps around midnight.
936 * <p>
937 * This instance is immutable and unaffected by this method call.
938 *
939 * @param nanos the nanos to add, may be negative
940 * @return an {@code OffsetTime} based on this time with the nanoseconds added, not null
```

```

941     */
942     public OffsetTime plusNanos(long nanos) {
943         return with(time.plusNanos(nanos), offset);
944     }
945
946     //-----
947     /**
948      * Returns a copy of this time with the specified amount subtracted.
949      * <p>
950      * This returns an {@code OffsetTime}, based on this one, with the specified amount subtracted.
951      * The amount is typically {@link Duration} but may be any other type implementing
952      * the {@link TemporalAmount} interface.
953      * <p>
954      * The calculation is delegated to the amount object by calling
955      * {@link TemporalAmount#subtractFrom(Temporal)}. The amount implementation is free
956      * to implement the subtraction in any way it wishes, however it typically
957      * calls back to {@link #minus(long, TemporalUnit)}. Consult the documentation
958      * of the amount implementation to determine if it can be successfully subtracted.
959      * <p>
960      * This instance is immutable and unaffected by this method call.
961      *
962      * @param amountToSubtract the amount to subtract, not null
963      * @return an {@code OffsetTime} based on this time with the subtraction made, not null
964      * @throws DateTimeException if the subtraction cannot be made
965      * @throws ArithmeticException if numeric overflow occurs
966      */
967     @Override
968     public OffsetTime minus(TemporalAmount amountToSubtract) {
969         return (OffsetTime) amountToSubtract.subtractFrom(this);
970     }
971
972     /**
973      * Returns a copy of this time with the specified amount subtracted.
974      * <p>
975      * This returns an {@code OffsetTime}, based on this one, with the amount
976      * in terms of the unit subtracted. If it is not possible to subtract the amount,
977      * because the unit is not supported or for some other reason, an exception is thrown.
978      * <p>
979      * This method is equivalent to {@link #plus(long, TemporalUnit)} with the amount negated.
980      * See that method for a full description of how addition, and thus subtraction, works.
981      * <p>
982      * This instance is immutable and unaffected by this method call.
983      *
984      * @param amountToSubtract the amount of the unit to subtract from the result, may be negative
985      * @param unit the unit of the amount to subtract, not null
986      * @return an {@code OffsetTime} based on this time with the specified amount subtracted, not
987      *         null
988      * @throws DateTimeException if the subtraction cannot be made
989      * @throws UnsupportedTemporalTypeException if the unit is not supported
990      * @throws ArithmeticException if numeric overflow occurs
991      */
992     @Override
993     public OffsetTime minus(long amountToSubtract, TemporalUnit unit) {

```

```

993         return (amountToSubtract == Long.MIN_VALUE ? plus(Long.MAX_VALUE, unit).plus(1, unit) :
994             plus(-amountToSubtract, unit));
995     }
996     //-----
997     /**
998      * Returns a copy of this {@code OffsetTime} with the specified number of hours subtracted.
999      * <p>
1000     * This subtracts the specified number of hours from this time, returning a new time.
1001     * The calculation wraps around midnight.
1002     * <p>
1003     * This instance is immutable and unaffected by this method call.
1004     *
1005     * @param hours    the hours to subtract, may be negative
1006     * @return an {@code OffsetTime} based on this time with the hours subtracted, not null
1007     */
1008     public OffsetTime minusHours(long hours) {
1009         return with(time.minusHours(hours), offset);
1010     }
1011
1012     /**
1013     * Returns a copy of this {@code OffsetTime} with the specified number of minutes subtracted.
1014     * <p>
1015     * This subtracts the specified number of minutes from this time, returning a new time.
1016     * The calculation wraps around midnight.
1017     * <p>
1018     * This instance is immutable and unaffected by this method call.
1019     *
1020     * @param minutes  the minutes to subtract, may be negative
1021     * @return an {@code OffsetTime} based on this time with the minutes subtracted, not null
1022     */
1023     public OffsetTime minusMinutes(long minutes) {
1024         return with(time.minusMinutes(minutes), offset);
1025     }
1026
1027     /**
1028     * Returns a copy of this {@code OffsetTime} with the specified number of seconds subtracted.
1029     * <p>
1030     * This subtracts the specified number of seconds from this time, returning a new time.
1031     * The calculation wraps around midnight.
1032     * <p>
1033     * This instance is immutable and unaffected by this method call.
1034     *
1035     * @param seconds  the seconds to subtract, may be negative
1036     * @return an {@code OffsetTime} based on this time with the seconds subtracted, not null
1037     */
1038     public OffsetTime minusSeconds(long seconds) {
1039         return with(time.minusSeconds(seconds), offset);
1040     }
1041
1042     /**
1043     * Returns a copy of this {@code OffsetTime} with the specified number of nanoseconds
1044     * subtracted.

```

```

1044 * <p>
1045 * This subtracts the specified number of nanoseconds from this time, returning a new time.
1046 * The calculation wraps around midnight.
1047 * <p>
1048 * This instance is immutable and unaffected by this method call.
1049 *
1050 * @param nanos the nanos to subtract, may be negative
1051 * @return an {@code OffsetTime} based on this time with the nanoseconds subtracted, not null
1052 */
1053 public OffsetTime minusNanos(long nanos) {
1054     return with(time.minusNanos(nanos), offset);
1055 }
1056
1057 //-----
1058 /**
1059 * Queries this time using the specified query.
1060 * <p>
1061 * This queries this time using the specified query strategy object.
1062 * The {@code TemporalQuery} object defines the logic to be used to
1063 * obtain the result. Read the documentation of the query to understand
1064 * what the result of this method will be.
1065 * <p>
1066 * The result of this method is obtained by invoking the
1067 * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
1068 * specified query passing {@code this} as the argument.
1069 *
1070 * @param <R> the type of the result
1071 * @param query the query to invoke, not null
1072 * @return the query result, null may be returned (defined by the query)
1073 * @throws DateTimeException if unable to query (defined by the query)
1074 * @throws ArithmeticException if numeric overflow occurs (defined by the query)
1075 */
1076 @SuppressWarnings("unchecked")
1077 @Override
1078 public <R> R query(TemporalQuery<R> query) {
1079     if (query == TemporalQueries.offset() || query == TemporalQueries.zone()) {
1080         return (R) offset;
1081     } else if (query == TemporalQueries.zoneId() | query == TemporalQueries.chronology() ||
1082         query == TemporalQueries.localDate()) {
1083         return null;
1084     } else if (query == TemporalQueries.localTime()) {
1085         return (R) time;
1086     } else if (query == TemporalQueries.precision()) {
1087         return (R) NANOS;
1088     }
1089     // inline TemporalAccessor.super.query(query) as an optimization
1090     // non-JDK classes are not permitted to make this optimization
1091     return query.queryFrom(this);
1092 }
1093
1094 /**
1095 * Adjusts the specified temporal object to have the same offset and time
1096 * as this object.

```

```

1096 * <p>
1097 * This returns a temporal object of the same observable type as the input
1098 * with the offset and time changed to be the same as this.
1099 * <p>
1100 * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
1101 * twice, passing {@link ChronoField#NANO_OF_DAY} and
1102 * {@link ChronoField#OFFSET_SECONDS} as the fields.
1103 * <p>
1104 * In most cases, it is clearer to reverse the calling pattern by using
1105 * {@link Temporal#with(TemporalAdjuster)}:
1106 * <pre>
1107 * // these two lines are equivalent, but the second approach is recommended
1108 * temporal = thisOffsetTime.adjustInto(temporal);
1109 * temporal = temporal.with(thisOffsetTime);
1110 * </pre>
1111 * <p>
1112 * This instance is immutable and unaffected by this method call.
1113 *
1114 * @param temporal the target object to be adjusted, not null
1115 * @return the adjusted object, not null
1116 * @throws DateTimeException if unable to make the adjustment
1117 * @throws ArithmeticException if numeric overflow occurs
1118 */
1119 @Override
1120 public Temporal adjustInto(Temporal temporal) {
1121     return temporal
1122         .with(NANO_OF_DAY, time.toNanoOfDay())
1123         .with(OFFSET_SECONDS, offset.getTotalSeconds());
1124 }
1125
1126 /**
1127 * Calculates the amount of time until another time in terms of the specified unit.
1128 * <p>
1129 * This calculates the amount of time between two {@code OffsetTime}
1130 * objects in terms of a single {@code TemporalUnit}.
1131 * The start and end points are {@code this} and the specified time.
1132 * The result will be negative if the end is before the start.
1133 * For example, the amount in hours between two times can be calculated
1134 * using {@code startTime.until(endTime, HOURS)}.
1135 * <p>
1136 * The {@code Temporal} passed to this method is converted to a
1137 * {@code OffsetTime} using {@link #from(TemporalAccessor)}.
1138 * If the offset differs between the two times, then the specified
1139 * end time is normalized to have the same offset as this time.
1140 * <p>
1141 * The calculation returns a whole number, representing the number of
1142 * complete units between the two times.
1143 * For example, the amount in hours between 11:30Z and 13:29Z will only
1144 * be one hour as it is one minute short of two hours.
1145 * <p>
1146 * There are two equivalent ways of using this method.
1147 * The first is to invoke this method.
1148 * The second is to use {@link TemporalUnit#between(Temporal, Temporal)}:

```

```

1149 * <pre>
1150 * // these two lines are equivalent
1151 * amount = start.until(end, MINUTES);
1152 * amount = MINUTES.between(start, end);
1153 * </pre>
1154 * The choice should be made based on which makes the code more readable.
1155 * <p>
1156 * The calculation is implemented in this method for {@link ChronoUnit}.
1157 * The units {@code NANOS}, {@code MICROS}, {@code MILLIS}, {@code SECONDS},
1158 * {@code MINUTES}, {@code HOURS} and {@code HALF_DAYS} are supported.
1159 * Other {@code ChronoUnit} values will throw an exception.
1160 * <p>
1161 * If the unit is not a {@code ChronoUnit}, then the result of this method
1162 * is obtained by invoking {@code TemporalUnit.between(Temporal, Temporal)}
1163 * passing {@code this} as the first argument and the converted input temporal
1164 * as the second argument.
1165 * <p>
1166 * This instance is immutable and unaffected by this method call.
1167 *
1168 * @param endExclusive the end time, exclusive, which is converted to an {@code OffsetTime},
1169 *    not null
1170 * @param unit the unit to measure the amount in, not null
1171 * @return the amount of time between this time and the end time
1172 * @throws DateTimeException if the amount cannot be calculated, or the end
1173 *    temporal cannot be converted to an {@code OffsetTime}
1174 * @throws UnsupportedTemporalTypeException if the unit is not supported
1175 * @throws ArithmeticException if numeric overflow occurs
1176 */
1177 @Override
1178 public long until(Temporal endExclusive, TemporalUnit unit) {
1179     OffsetTime end = OffsetTime.from(endExclusive);
1180     if (unit instanceof ChronoUnit) {
1181         long nanosUntil = end.toEpochNano() - toEpochNano(); // no overflow
1182         switch ((ChronoUnit) unit) {
1183             case NANOS: return nanosUntil;
1184             case MICROS: return nanosUntil / 1000;
1185             case MILLIS: return nanosUntil / 1000_000;
1186             case SECONDS: return nanosUntil / NANOS_PER_SECOND;
1187             case MINUTES: return nanosUntil / NANOS_PER_MINUTE;
1188             case HOURS: return nanosUntil / NANOS_PER_HOUR;
1189             case HALF_DAYS: return nanosUntil / (12 * NANOS_PER_HOUR);
1190         }
1191         throw new UnsupportedTemporalTypeException("Unsupported unit: " + unit);
1192     }
1193     return unit.between(this, end);
1194 }
1195 /**
1196 * Formats this time using the specified formatter.
1197 * <p>
1198 * This time will be passed to the formatter to produce a string.
1199 *
1200 * @param formatter the formatter to use, not null

```

```

1201     * @return the formatted time string, not null
1202     * @throws DateTimeException if an error occurs during printing
1203     */
1204     public String format(DateTimeFormatter formatter) {
1205         Objects.requireNonNull(formatter, "formatter");
1206         return formatter.format(this);
1207     }
1208
1209     //-----
1210     /**
1211      * Combines this time with a date to create an {@code OffsetDateTime}.
1212      * <p>
1213      * This returns an {@code OffsetDateTime} formed from this time and the specified date.
1214      * All possible combinations of date and time are valid.
1215      *
1216      * @param date the date to combine with, not null
1217      * @return the offset date-time formed from this time and the specified date, not null
1218      */
1219     public OffsetDateTime atDate(LocalDate date) {
1220         return OffsetDateTime.of(date, time, offset);
1221     }
1222
1223     //-----
1224     /**
1225      * Converts this time to epoch nanos based on 1970-01-01Z.
1226      *
1227      * @return the epoch nanos value
1228      */
1229     private long toEpochNano() {
1230         long nod = time.toNanoOfDay();
1231         long offsetNanos = offset.getTotalSeconds() * NANOS_PER_SECOND;
1232         return nod - offsetNanos;
1233     }
1234
1235     //-----
1236     /**
1237      * Compares this {@code OffsetTime} to another time.
1238      * <p>
1239      * The comparison is based first on the UTC equivalent instant, then on the local time.
1240      * It is "consistent with equals", as defined by {@link Comparable}.
1241      * <p>
1242      * For example, the following is the comparator order:
1243      * <ol>
1244      * <li>{@code 10:30+01:00}</li>
1245      * <li>{@code 11:00+01:00}</li>
1246      * <li>{@code 12:00+02:00}</li>
1247      * <li>{@code 11:30+01:00}</li>
1248      * <li>{@code 12:00+01:00}</li>
1249      * <li>{@code 12:30+01:00}</li>
1250      * </ol>
1251      * Values #2 and #3 represent the same instant on the time-line.
1252      * When two values represent the same instant, the local time is compared
1253      * to distinguish them. This step is needed to make the ordering

```

```

1254     * consistent with {@code equals()}.
1255     * <p>
1256     * To compare the underlying local time of two {@code TemporalAccessor} instances,
1257     * use {@link ChronoField#NANO_OF_DAY} as a comparator.
1258     *
1259     * @param other the other time to compare to, not null
1260     * @return the comparator value, negative if less, positive if greater
1261     * @throws NullPointerException if {@code other} is null
1262     */
1263     @Override
1264     public int compareTo(OffsetTime other) {
1265         if (offset.equals(other.offset)) {
1266             return time.compareTo(other.time);
1267         }
1268         int compare = Long.compare(toEpochNano(), other.toEpochNano());
1269         if (compare == 0) {
1270             compare = time.compareTo(other.time);
1271         }
1272         return compare;
1273     }
1274
1275     //-----
1276     /**
1277      * Checks if the instant of this {@code OffsetTime} is after that of the
1278      * specified time applying both times to a common date.
1279      * <p>
1280      * This method differs from the comparison in {@link #compareTo} in that it
1281      * only compares the instant of the time. This is equivalent to converting both
1282      * times to an instant using the same date and comparing the instants.
1283      *
1284      * @param other the other time to compare to, not null
1285      * @return true if this is after the instant of the specified time
1286      */
1287     public boolean isAfter(OffsetTime other) {
1288         return toEpochNano() > other.toEpochNano();
1289     }
1290
1291     /**
1292      * Checks if the instant of this {@code OffsetTime} is before that of the
1293      * specified time applying both times to a common date.
1294      * <p>
1295      * This method differs from the comparison in {@link #compareTo} in that it
1296      * only compares the instant of the time. This is equivalent to converting both
1297      * times to an instant using the same date and comparing the instants.
1298      *
1299      * @param other the other time to compare to, not null
1300      * @return true if this is before the instant of the specified time
1301      */
1302     public boolean isBefore(OffsetTime other) {
1303         return toEpochNano() < other.toEpochNano();
1304     }
1305
1306     /**

```



```

1307     * Checks if the instant of this {@code OffsetTime} is equal to that of the
1308     * specified time applying both times to a common date.
1309     * <p>
1310     * This method differs from the comparison in {@link #compareTo} and {@link #equals}
1311     * in that it only compares the instant of the time. This is equivalent to converting both
1312     * times to an instant using the same date and comparing the instants.
1313     *
1314     * @param other the other time to compare to, not null
1315     * @return true if this is equal to the instant of the specified time
1316     */
1317     public boolean isEqual(OffsetTime other) {
1318         return toEpochNano() == other.toEpochNano();
1319     }
1320
1321     //-----
1322     /**
1323     * Checks if this time is equal to another time.
1324     * <p>
1325     * The comparison is based on the local-time and the offset.
1326     * To compare for the same instant on the time-line, use {@link #isEqual(OffsetTime)}.
1327     * <p>
1328     * Only objects of type {@code OffsetTime} are compared, other types return false.
1329     * To compare the underlying local time of two {@code TemporalAccessor} instances,
1330     * use {@link ChronoField#NANO_OF_DAY} as a comparator.
1331     *
1332     * @param obj the object to check, null returns false
1333     * @return true if this is equal to the other time
1334     */
1335     @Override
1336     public boolean equals(Object obj) {
1337         if (this == obj) {
1338             return true;
1339         }
1340         if (obj instanceof OffsetTime) {
1341             OffsetTime other = (OffsetTime) obj;
1342             return time.equals(other.time) && offset.equals(other.offset);
1343         }
1344         return false;
1345     }
1346
1347     /**
1348     * A hash code for this time.
1349     *
1350     * @return a suitable hash code
1351     */
1352     @Override
1353     public int hashCode() {
1354         return time.hashCode() ^ offset.hashCode();
1355     }
1356
1357     //-----
1358     /**
1359     * Outputs this time as a {@code String}, such as {@code 10:15:30+01:00}.

```

```

1360 * <p>
1361 * The output will be one of the following ISO-8601 formats:
1362 * <ul>
1363 * <li>{@code HH:mmXXXXX}</li>
1364 * <li>{@code HH:mm:ssXXXXX}</li>
1365 * <li>{@code HH:mm:ss.SSSXXXXX}</li>
1366 * <li>{@code HH:mm:ss.SSSSSXXXXX}</li>
1367 * <li>{@code HH:mm:ss.SSSSSSSXXXXX}</li>
1368 * </ul>
1369 * The format used will be the shortest that outputs the full value of
1370 * the time where the omitted parts are implied to be zero.
1371 *
1372 * @return a string representation of this time, not null
1373 */
1374 @Override
1375 public String toString() {
1376     return time.toString() + offset.toString();
1377 }
1378
1379 //-----
1380 /**
1381 * Writes the object using a
1382 * <a href="../../serialized-form.html#java.time.Ser">dedicated serialized form</a>.
1383 * @serialData
1384 * <pre>
1385 *   out.writeByte(9); // identifies an OffsetTime
1386 *   // the <a href="../../serialized-form.html#java.time.LocalTime">time</a> excluding the one
1387 *       byte header
1388 *   // the <a href="../../serialized-form.html#java.time.ZoneOffset">offset</a> excluding the
1389 *       one byte header
1390 * </pre>
1391 *
1392 * @return the instance of {@code Ser}, not null
1393 */
1394 private Object writeReplace() {
1395     return new Ser(Ser.OFFSET_TIME_TYPE, this);
1396 }
1397
1398 /**
1399 * Defend against malicious streams.
1400 *
1401 * @param s the stream to read
1402 * @throws InvalidObjectException always
1403 */
1404 private void readObject(ObjectInputStream s) throws InvalidObjectException {
1405     throw new InvalidObjectException("Deserialization via serialization delegate");
1406 }
1407
1408 void writeExternal(ObjectOutput out) throws IOException {
1409     time.writeExternal(out);
1410     offset.writeExternal(out);
1411 }

```

```

1411     static OffsetTime readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
1412         LocalTime time = LocalTime.readExternal(in);
1413         ZoneOffset offset = ZoneOffset.readExternal(in);
1414         return OffsetTime.of(time, offset);
1415     }
1416 }
1417 }

```

4.13 Period.java

Secuencia 62: java/time/Period.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2008-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,

```

```
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
62 package java.time;
63
64 import static java.time.temporal.ChronoUnit.DAYS;
65 import static java.time.temporal.ChronoUnit.MONTHS;
66 import static java.time.temporal.ChronoUnit.YEARS;
67
68 import java.io.DataInput;
69 import java.io.DataOutput;
70 import java.io.IOException;
71 import java.io.InvalidObjectException;
72 import java.io.ObjectInputStream;
73 import java.io.Serializable;
74 import java.time.chrono.ChronoLocalDate;
75 import java.time.chrono.ChronoPeriod;
76 import java.time.chrono.Chronology;
77 import java.time.chrono.IsoChronology;
78 import java.time.format.DateTimeParseException;
79 import java.time.temporal.ChronoUnit;
80 import java.time.temporal.Temporal;
81 import java.time.temporal.TemporalAccessor;
82 import java.time.temporal.TemporalAmount;
83 import java.time.temporal.TemporalQueries;
84 import java.time.temporal.TemporalUnit;
85 import java.time.temporal.UnsupportedTemporalTypeException;
86 import java.util.Arrays;
87 import java.util.Collections;
88 import java.util.List;
89 import java.util.Objects;
90 import java.util.regex.Matcher;
91 import java.util.regex.Pattern;
92
93 /**
94 * A date-based amount of time in the ISO-8601 calendar system,
95 * such as '2 years, 3 months and 4 days'.
```

```

96  * <p>
97  * This class models a quantity or amount of time in terms of years, months and days.
98  * See {@link Duration} for the time-based equivalent to this class.
99  * <p>
100 * Durations and periods differ in their treatment of daylight savings time
101 * when added to {@link ZonedDateTime}. A {@code Duration} will add an exact
102 * number of seconds, thus a duration of one day is always exactly 24 hours.
103 * By contrast, a {@code Period} will add a conceptual day, trying to maintain
104 * the local time.
105 * <p>
106 * For example, consider adding a period of one day and a duration of one day to
107 * 18:00 on the evening before a daylight savings gap. The {@code Period} will add
108 * the conceptual day and result in a {@code ZonedDateTime} at 18:00 the following day.
109 * By contrast, the {@code Duration} will add exactly 24 hours, resulting in a
110 * {@code ZonedDateTime} at 19:00 the following day (assuming a one hour DST gap).
111 * <p>
112 * The supported units of a period are {@link ChronoUnit#YEARS YEARS},
113 * {@link ChronoUnit#MONTHS MONTHS} and {@link ChronoUnit#DAYS DAYS}.
114 * All three fields are always present, but may be set to zero.
115 * <p>
116 * The ISO-8601 calendar system is the modern civil calendar system used today
117 * in most of the world. It is equivalent to the proleptic Gregorian calendar
118 * system, in which today's rules for leap years are applied for all time.
119 * <p>
120 * The period is modeled as a directed amount of time, meaning that individual parts of the
121 * period may be negative.
122 *
123 * <p>
124 * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
125 * class; use of identity-sensitive operations (including reference equality
126 * ({@code ==}), identity hash code, or synchronization) on instances of
127 * {@code Period} may have unpredictable results and should be avoided.
128 * The {@code equals} method should be used for comparisons.
129 *
130 * @implSpec
131 * This class is immutable and thread-safe.
132 *
133 * @since 1.8
134 */
135 public final class Period
136     implements ChronoPeriod, Serializable {
137
138     /**
139      * A constant for a period of zero.
140      */
141     public static final Period ZERO = new Period(0, 0, 0);
142
143     /**
144      * Serialization version.
145      */
146     private static final long serialVersionUID = -3587258372562876L;
147
148     /**
149      * The pattern for parsing.
150      */

```

```

149 private static final Pattern PATTERN =
150     Pattern.compile("([-+]?)P(?:([-+]?[0-9]+)Y)?(?:([-+]?[0-9]+)M)?(?:([-+]?[0-9]+)W)
151         ?(?:([-+]?[0-9]+)D)?", Pattern.CASE_INSENSITIVE);
152
153 /**
154  * The set of supported units.
155  */
156 private static final List<TemporalUnit> SUPPORTED_UNITS =
157     Collections.unmodifiableList(Arrays.<TemporalUnit>asList(YEARS, MONTHS, DAYS));
158
159 /**
160  * The number of years.
161  */
162 private final int years;
163 /**
164  * The number of months.
165  */
166 private final int months;
167 /**
168  * The number of days.
169  */
170 private final int days;
171
172 //-----
173 /**
174  * Obtains a {@code Period} representing a number of years.
175  * <p>
176  * The resulting period will have the specified years.
177  * The months and days units will be zero.
178  *
179  * @param years the number of years, positive or negative
180  * @return the period of years, not null
181  */
182 public static Period ofYears(int years) {
183     return create(years, 0, 0);
184 }
185
186 /**
187  * Obtains a {@code Period} representing a number of months.
188  * <p>
189  * The resulting period will have the specified months.
190  * The years and days units will be zero.
191  *
192  * @param months the number of months, positive or negative
193  * @return the period of months, not null
194  */
195 public static Period ofMonths(int months) {
196     return create(0, months, 0);
197 }
198
199 /**
200  * Obtains a {@code Period} representing a number of weeks.
201  * <p>

```

```

201     * The resulting period will be day-based, with the amount of days
202     * equal to the number of weeks multiplied by 7.
203     * The years and months units will be zero.
204     *
205     * @param weeks the number of weeks, positive or negative
206     * @return the period, with the input weeks converted to days, not null
207     */
208     public static Period ofWeeks(int weeks) {
209         return create(0, 0, Math.multiplyExact(weeks, 7));
210     }
211
212     /**
213     * Obtains a {@code Period} representing a number of days.
214     * <p>
215     * The resulting period will have the specified days.
216     * The years and months units will be zero.
217     *
218     * @param days the number of days, positive or negative
219     * @return the period of days, not null
220     */
221     public static Period ofDays(int days) {
222         return create(0, 0, days);
223     }
224
225     //-----
226     /**
227     * Obtains a {@code Period} representing a number of years, months and days.
228     * <p>
229     * This creates an instance based on years, months and days.
230     *
231     * @param years the amount of years, may be negative
232     * @param months the amount of months, may be negative
233     * @param days the amount of days, may be negative
234     * @return the period of years, months and days, not null
235     */
236     public static Period of(int years, int months, int days) {
237         return create(years, months, days);
238     }
239
240     //-----
241     /**
242     * Obtains an instance of {@code Period} from a temporal amount.
243     * <p>
244     * This obtains a period based on the specified amount.
245     * A {@code TemporalAmount} represents an amount of time, which may be
246     * date-based or time-based, which this factory extracts to a {@code Period}.
247     * <p>
248     * The conversion loops around the set of units from the amount and uses
249     * the {@link ChronoUnit#YEARS YEARS}, {@link ChronoUnit#MONTHS MONTHS}
250     * and {@link ChronoUnit#DAYS DAYS} units to create a period.
251     * If any other units are found then an exception is thrown.
252     * <p>
253     * If the amount is a {@code ChronoPeriod} then it must use the ISO chronology.

```

```

254 *
255 * @param amount the temporal amount to convert, not null
256 * @return the equivalent period, not null
257 * @throws DateTimeException if unable to convert to a {@code Period}
258 * @throws ArithmeticException if the amount of years, months or days exceeds an int
259 */
260 public static Period from(TemporalAmount amount) {
261     if (amount instanceof Period) {
262         return (Period) amount;
263     }
264     if (amount instanceof ChronoPeriod) {
265         if (IsoChronology.INSTANCE.equals(((ChronoPeriod) amount).getChronology()) == false) {
266             throw new DateTimeException("Period requires ISO chronology: " + amount);
267         }
268     }
269     Objects.requireNonNull(amount, "amount");
270     int years = 0;
271     int months = 0;
272     int days = 0;
273     for (TemporalUnit unit : amount.getUnits()) {
274         long unitAmount = amount.get(unit);
275         if (unit == ChronoUnit.YEARS) {
276             years = Math.toIntExact(unitAmount);
277         } else if (unit == ChronoUnit.MONTHS) {
278             months = Math.toIntExact(unitAmount);
279         } else if (unit == ChronoUnit.DAYS) {
280             days = Math.toIntExact(unitAmount);
281         } else {
282             throw new DateTimeException("Unit must be Years, Months or Days, but was " + unit);
283         }
284     }
285     return create(years, months, days);
286 }
287
288 //-----
289 /**
290  * Obtains a {@code Period} from a text string such as {@code PnYnMnD}.
291  * <p>
292  * This will parse the string produced by {@code toString()} which is
293  * based on the ISO-8601 period formats {@code PnYnMnD} and {@code PnW}.
294  * <p>
295  * The string starts with an optional sign, denoted by the ASCII negative
296  * or positive symbol. If negative, the whole period is negated.
297  * The ASCII letter "P" is next in upper or lower case.
298  * There are then four sections, each consisting of a number and a suffix.
299  * At least one of the four sections must be present.
300  * The sections have suffixes in ASCII of "Y", "M", "W" and "D" for
301  * years, months, weeks and days, accepted in upper or lower case.
302  * The suffixes must occur in order.
303  * The number part of each section must consist of ASCII digits.
304  * The number may be prefixed by the ASCII negative or positive symbol.
305  * The number must parse to an {@code int}.
306  * <p>

```



```

307 * The leading plus/minus sign, and negative values for other units are
308 * not part of the ISO-8601 standard. In addition, ISO-8601 does not
309 * permit mixing between the {@code PnYnMnD} and {@code PnW} formats.
310 * Any week-based input is multiplied by 7 and treated as a number of days.
311 * <p>
312 * For example, the following are valid inputs:
313 * <pre>
314 * "P2Y"          -- Period.ofYears(2)
315 * "P3M"          -- Period.ofMonths(3)
316 * "P4W"          -- Period.ofWeeks(4)
317 * "P5D"          -- Period.ofDays(5)
318 * "P1Y2M3D"      -- Period.of(1, 2, 3)
319 * "P1Y2M3W4D"    -- Period.of(1, 2, 25)
320 * "P-1Y2M"       -- Period.of(-1, 2, 0)
321 * "-P1Y2M"       -- Period.of(-1, -2, 0)
322 * </pre>
323 *
324 * @param text the text to parse, not null
325 * @return the parsed period, not null
326 * @throws DateTimeParseException if the text cannot be parsed to a period
327 */
328 public static Period parse(CharSequence text) {
329     Objects.requireNonNull(text, "text");
330     Matcher matcher = PATTERN.matcher(text);
331     if (matcher.matches()) {
332         int negate = (".".equals(matcher.group(1)) ? -1 : 1);
333         String yearMatch = matcher.group(2);
334         String monthMatch = matcher.group(3);
335         String weekMatch = matcher.group(4);
336         String dayMatch = matcher.group(5);
337         if (yearMatch != null || monthMatch != null || dayMatch != null || weekMatch != null) {
338             try {
339                 int years = parseNumber(text, yearMatch, negate);
340                 int months = parseNumber(text, monthMatch, negate);
341                 int weeks = parseNumber(text, weekMatch, negate);
342                 int days = parseNumber(text, dayMatch, negate);
343                 days = Math.addExact(days, Math.multiplyExact(weeks, 7));
344                 return create(years, months, days);
345             } catch (NumberFormatException ex) {
346                 throw new DateTimeParseException("Text cannot be parsed to a Period", text, 0,
347                     ex);
348             }
349         }
350         throw new DateTimeParseException("Text cannot be parsed to a Period", text, 0);
351     }
352 }
353 private static int parseNumber(CharSequence text, String str, int negate) {
354     if (str == null) {
355         return 0;
356     }
357     int val = Integer.parseInt(str);
358     try {

```

```

359         return Math.multiplyExact(val, negate);
360     } catch (ArithmeticException ex) {
361         throw new DateTimeParseException("Text cannot be parsed to a Period", text, 0, ex);
362     }
363 }
364
365 //-----
366 /**
367  * Obtains a {@code Period} consisting of the number of years, months,
368  * and days between two dates.
369  * <p>
370  * The start date is included, but the end date is not.
371  * The period is calculated by removing complete months, then calculating
372  * the remaining number of days, adjusting to ensure that both have the same sign.
373  * The number of months is then split into years and months based on a 12 month year.
374  * A month is considered if the end day-of-month is greater than or equal to the start day-of-
375     month.
376     * For example, from {@code 2010-01-15} to {@code 2011-03-18} is one year, two months and three
377     days.
378     * <p>
379     * The result of this method can be a negative period if the end is before the start.
380     * The negative sign will be the same in each of year, month and day.
381     *
382     * @param startDateInclusive the start date, inclusive, not null
383     * @param endDateExclusive the end date, exclusive, not null
384     * @return the period between this date and the end date, not null
385     * @see ChronoLocalDate#until(ChronoLocalDate)
386     */
387 public static Period between(LocalDate startDateInclusive, LocalDate endDateExclusive) {
388     return startDateInclusive.until(endDateExclusive);
389 }
390
391 //-----
392 /**
393  * Creates an instance.
394  *
395  * @param years the amount
396  * @param months the amount
397  * @param days the amount
398  */
399 private static Period create(int years, int months, int days) {
400     if ((years | months | days) == 0) {
401         return ZERO;
402     }
403     return new Period(years, months, days);
404 }
405
406 /**
407  * Constructor.
408  *
409  * @param years the amount
410  * @param months the amount
411  * @param days the amount

```

```

410     */
411     private Period(int years, int months, int days) {
412         this.years = years;
413         this.months = months;
414         this.days = days;
415     }
416
417     //-----
418     /**
419      * Gets the value of the requested unit.
420      * <p>
421      * This returns a value for each of the three supported units,
422      * {@link ChronoUnit#YEARS YEARS}, {@link ChronoUnit#MONTHS MONTHS} and
423      * {@link ChronoUnit#DAYS DAYS}.
424      * All other units throw an exception.
425      *
426      * @param unit the {@code TemporalUnit} for which to return the value
427      * @return the long value of the unit
428      * @throws DateTimeException if the unit is not supported
429      * @throws UnsupportedTemporalTypeException if the unit is not supported
430      */
431     @Override
432     public long get(TemporalUnit unit) {
433         if (unit == ChronoUnit.YEARS) {
434             return getYears();
435         } else if (unit == ChronoUnit.MONTHS) {
436             return getMonths();
437         } else if (unit == ChronoUnit.DAYS) {
438             return getDays();
439         } else {
440             throw new UnsupportedTemporalTypeException("Unsupported unit: " + unit);
441         }
442     }
443
444     /**
445      * Gets the set of units supported by this period.
446      * <p>
447      * The supported units are {@link ChronoUnit#YEARS YEARS},
448      * {@link ChronoUnit#MONTHS MONTHS} and {@link ChronoUnit#DAYS DAYS}.
449      * They are returned in the order years, months, days.
450      * <p>
451      * This set can be used in conjunction with {@link #get(TemporalUnit)}
452      * to access the entire state of the period.
453      *
454      * @return a list containing the years, months and days units, not null
455      */
456     @Override
457     public List<TemporalUnit> getUnits() {
458         return SUPPORTED_UNITS;
459     }
460
461     /**
462      * Gets the chronology of this period, which is the ISO calendar system.

```

```

463     * <p>
464     * The {@code Chronology} represents the calendar system in use.
465     * The ISO-8601 calendar system is the modern civil calendar system used today
466     * in most of the world. It is equivalent to the proleptic Gregorian calendar
467     * system, in which today's rules for leap years are applied for all time.
468     *
469     * @return the ISO chronology, not null
470     */
471     @Override
472     public IsoChronology getChronology() {
473         return IsoChronology.INSTANCE;
474     }
475
476     //-----
477     /**
478     * Checks if all three units of this period are zero.
479     * <p>
480     * A zero period has the value zero for the years, months and days units.
481     *
482     * @return true if this period is zero-length
483     */
484     public boolean isZero() {
485         return (this == ZERO);
486     }
487
488     /**
489     * Checks if any of the three units of this period are negative.
490     * <p>
491     * This checks whether the years, months or days units are less than zero.
492     *
493     * @return true if any unit of this period is negative
494     */
495     public boolean isNegative() {
496         return years < 0 || months < 0 || days < 0;
497     }
498
499     //-----
500     /**
501     * Gets the amount of years of this period.
502     * <p>
503     * This returns the years unit.
504     * <p>
505     * The months unit is not automatically normalized with the years unit.
506     * This means that a period of "15 months" is different to a period
507     * of "1 year and 3 months".
508     *
509     * @return the amount of years of this period, may be negative
510     */
511     public int getYears() {
512         return years;
513     }
514
515     /**

```

```
516     * Gets the amount of months of this period.
517     * <p>
518     * This returns the months unit.
519     * <p>
520     * The months unit is not automatically normalized with the years unit.
521     * This means that a period of "15 months" is different to a period
522     * of "1 year and 3 months".
523     *
524     * @return the amount of months of this period, may be negative
525     */
526     public int getMonths() {
527         return months;
528     }
529
530     /**
531     * Gets the amount of days of this period.
532     * <p>
533     * This returns the days unit.
534     *
535     * @return the amount of days of this period, may be negative
536     */
537     public int getDays() {
538         return days;
539     }
540
541     //-----
542     /**
543     * Returns a copy of this period with the specified amount of years.
544     * <p>
545     * This sets the amount of the years unit in a copy of this period.
546     * The months and days units are unaffected.
547     * <p>
548     * The months unit is not automatically normalized with the years unit.
549     * This means that a period of "15 months" is different to a period
550     * of "1 year and 3 months".
551     * <p>
552     * This instance is immutable and unaffected by this method call.
553     *
554     * @param years  the years to represent, may be negative
555     * @return a {@code Period} based on this period with the requested years, not null
556     */
557     public Period withYears(int years) {
558         if (years == this.years) {
559             return this;
560         }
561         return create(years, months, days);
562     }
563
564     /**
565     * Returns a copy of this period with the specified amount of months.
566     * <p>
567     * This sets the amount of the months unit in a copy of this period.
568     * The years and days units are unaffected.
```

```

569     * <p>
570     * The months unit is not automatically normalized with the years unit.
571     * This means that a period of "15 months" is different to a period
572     * of "1 year and 3 months".
573     * <p>
574     * This instance is immutable and unaffected by this method call.
575     *
576     * @param months the months to represent, may be negative
577     * @return a {@code Period} based on this period with the requested months, not null
578     */
579     public Period withMonths(int months) {
580         if (months == this.months) {
581             return this;
582         }
583         return create(years, months, days);
584     }
585
586     /**
587     * Returns a copy of this period with the specified amount of days.
588     * <p>
589     * This sets the amount of the days unit in a copy of this period.
590     * The years and months units are unaffected.
591     * <p>
592     * This instance is immutable and unaffected by this method call.
593     *
594     * @param days the days to represent, may be negative
595     * @return a {@code Period} based on this period with the requested days, not null
596     */
597     public Period withDays(int days) {
598         if (days == this.days) {
599             return this;
600         }
601         return create(years, months, days);
602     }
603
604     //-----
605     /**
606     * Returns a copy of this period with the specified period added.
607     * <p>
608     * This operates separately on the years, months and days.
609     * No normalization is performed.
610     * <p>
611     * For example, "1 year, 6 months and 3 days" plus "2 years, 2 months and 2 days"
612     * returns "3 years, 8 months and 5 days".
613     * <p>
614     * The specified amount is typically an instance of {@code Period}.
615     * Other types are interpreted using {@link Period#from(TemporalAmount)}.
616     * <p>
617     * This instance is immutable and unaffected by this method call.
618     *
619     * @param amountToAdd the amount to add, not null
620     * @return a {@code Period} based on this period with the requested period added, not null
621     * @throws DateTimeException if the specified amount has a non-ISO chronology or

```

```

622     * contains an invalid unit
623     * @throws ArithmeticException if numeric overflow occurs
624     */
625     public Period plus(TemporalAmount amountToAdd) {
626         Period isoAmount = Period.from(amountToAdd);
627         return create(
628             Math.addExact(years, isoAmount.years),
629             Math.addExact(months, isoAmount.months),
630             Math.addExact(days, isoAmount.days));
631     }
632
633     /**
634     * Returns a copy of this period with the specified years added.
635     * <p>
636     * This adds the amount to the years unit in a copy of this period.
637     * The months and days units are unaffected.
638     * For example, "1 year, 6 months and 3 days" plus 2 years returns "3 years, 6 months and 3
        days".
639     * <p>
640     * This instance is immutable and unaffected by this method call.
641     *
642     * @param yearsToAdd the years to add, positive or negative
643     * @return a {@code Period} based on this period with the specified years added, not null
644     * @throws ArithmeticException if numeric overflow occurs
645     */
646     public Period plusYears(long yearsToAdd) {
647         if (yearsToAdd == 0) {
648             return this;
649         }
650         return create(Math.toIntExact(Math.addExact(years, yearsToAdd)), months, days);
651     }
652
653     /**
654     * Returns a copy of this period with the specified months added.
655     * <p>
656     * This adds the amount to the months unit in a copy of this period.
657     * The years and days units are unaffected.
658     * For example, "1 year, 6 months and 3 days" plus 2 months returns "1 year, 8 months and 3
        days".
659     * <p>
660     * This instance is immutable and unaffected by this method call.
661     *
662     * @param monthsToAdd the months to add, positive or negative
663     * @return a {@code Period} based on this period with the specified months added, not null
664     * @throws ArithmeticException if numeric overflow occurs
665     */
666     public Period plusMonths(long monthsToAdd) {
667         if (monthsToAdd == 0) {
668             return this;
669         }
670         return create(years, Math.toIntExact(Math.addExact(months, monthsToAdd)), days);
671     }
672

```

```

673 /**
674  * Returns a copy of this period with the specified days added.
675  * <p>
676  * This adds the amount to the days unit in a copy of this period.
677  * The years and months units are unaffected.
678  * For example, "1 year, 6 months and 3 days" plus 2 days returns "1 year, 6 months and 5 days
679  * ".
680  * <p>
681  * This instance is immutable and unaffected by this method call.
682  *
683  * @param daysToAdd the days to add, positive or negative
684  * @return a {@code Period} based on this period with the specified days added, not null
685  * @throws ArithmeticException if numeric overflow occurs
686  */
687 public Period plusDays(long daysToAdd) {
688     if (daysToAdd == 0) {
689         return this;
690     }
691     return create(years, months, Math.toIntExact(Math.addExact(days, daysToAdd)));
692 }
693 //-----
694 /**
695  * Returns a copy of this period with the specified period subtracted.
696  * <p>
697  * This operates separately on the years, months and days.
698  * No normalization is performed.
699  * <p>
700  * For example, "1 year, 6 months and 3 days" minus "2 years, 2 months and 2 days"
701  * returns "-1 years, 4 months and 1 day".
702  * <p>
703  * The specified amount is typically an instance of {@code Period}.
704  * Other types are interpreted using {@link Period#from(TemporalAmount)}.
705  * <p>
706  * This instance is immutable and unaffected by this method call.
707  *
708  * @param amountToSubtract the amount to subtract, not null
709  * @return a {@code Period} based on this period with the requested period subtracted, not null
710  * @throws DateTimeException if the specified amount has a non-ISO chronology or
711  *         contains an invalid unit
712  * @throws ArithmeticException if numeric overflow occurs
713  */
714 public Period minus(TemporalAmount amountToSubtract) {
715     Period isoAmount = Period.from(amountToSubtract);
716     return create(
717         Math.subtractExact(years, isoAmount.years),
718         Math.subtractExact(months, isoAmount.months),
719         Math.subtractExact(days, isoAmount.days));
720 }
721
722 /**
723  * Returns a copy of this period with the specified years subtracted.
724  * <p>

```



```

725     * This subtracts the amount from the years unit in a copy of this period.
726     * The months and days units are unaffected.
727     * For example, "1 year, 6 months and 3 days" minus 2 years returns "-1 years, 6 months and 3
       days".
728     * <p>
729     * This instance is immutable and unaffected by this method call.
730     *
731     * @param yearsToSubtract the years to subtract, positive or negative
732     * @return a {@code Period} based on this period with the specified years subtracted, not null
733     * @throws ArithmeticException if numeric overflow occurs
734     */
735     public Period minusYears(long yearsToSubtract) {
736         return (yearsToSubtract == Long.MIN_VALUE ? plusYears(Long.MAX_VALUE).plusYears(1) :
           plusYears(-yearsToSubtract));
737     }
738
739     /**
740     * Returns a copy of this period with the specified months subtracted.
741     * <p>
742     * This subtracts the amount from the months unit in a copy of this period.
743     * The years and days units are unaffected.
744     * For example, "1 year, 6 months and 3 days" minus 2 months returns "1 year, 4 months and 3
       days".
745     * <p>
746     * This instance is immutable and unaffected by this method call.
747     *
748     * @param monthsToSubtract the years to subtract, positive or negative
749     * @return a {@code Period} based on this period with the specified months subtracted, not null
750     * @throws ArithmeticException if numeric overflow occurs
751     */
752     public Period minusMonths(long monthsToSubtract) {
753         return (monthsToSubtract == Long.MIN_VALUE ? plusMonths(Long.MAX_VALUE).plusMonths(1) :
           plusMonths(-monthsToSubtract));
754     }
755
756     /**
757     * Returns a copy of this period with the specified days subtracted.
758     * <p>
759     * This subtracts the amount from the days unit in a copy of this period.
760     * The years and months units are unaffected.
761     * For example, "1 year, 6 months and 3 days" minus 2 days returns "1 year, 6 months and 1 day
       ".
762     * <p>
763     * This instance is immutable and unaffected by this method call.
764     *
765     * @param daysToSubtract the months to subtract, positive or negative
766     * @return a {@code Period} based on this period with the specified days subtracted, not null
767     * @throws ArithmeticException if numeric overflow occurs
768     */
769     public Period minusDays(long daysToSubtract) {
770         return (daysToSubtract == Long.MIN_VALUE ? plusDays(Long.MAX_VALUE).plusDays(1) : plusDays
           (-daysToSubtract));
771     }

```

```

772 //-----
773 /**
774  * Returns a new instance with each element in this period multiplied
775  * by the specified scalar.
776  * <p>
777  * This returns a period with each of the years, months and days units
778  * individually multiplied.
779  * For example, a period of "2 years, -3 months and 4 days" multiplied by
780  * 3 will return "6 years, -9 months and 12 days".
781  * No normalization is performed.
782  *
783  * @param scalar the scalar to multiply by, not null
784  * @return a {@code Period} based on this period with the amounts multiplied by the scalar, not
785  *         null
786  * @throws ArithmeticException if numeric overflow occurs
787  */
788 public Period multipliedBy(int scalar) {
789     if (this == ZERO || scalar == 1) {
790         return this;
791     }
792     return create(
793         Math.multiplyExact(years, scalar),
794         Math.multiplyExact(months, scalar),
795         Math.multiplyExact(days, scalar));
796 }
797
798 /**
799  * Returns a new instance with each amount in this period negated.
800  * <p>
801  * This returns a period with each of the years, months and days units
802  * individually negated.
803  * For example, a period of "2 years, -3 months and 4 days" will be
804  * negated to "-2 years, 3 months and -4 days".
805  * No normalization is performed.
806  *
807  * @return a {@code Period} based on this period with the amounts negated, not null
808  * @throws ArithmeticException if numeric overflow occurs, which only happens if
809  *         one of the units has the value {@code Long.MIN_VALUE}
810  */
811 public Period negated() {
812     return multipliedBy(-1);
813 }
814
815 //-----
816 /**
817  * Returns a copy of this period with the years and months normalized.
818  * <p>
819  * This normalizes the years and months units, leaving the days unit unchanged.
820  * The months unit is adjusted to have an absolute value less than 11,
821  * with the years unit being adjusted to compensate. For example, a period of
822  * "1 Year and 15 months" will be normalized to "2 years and 3 months".
823  * <p>

```

```

824     * The sign of the years and months units will be the same after normalization.
825     * For example, a period of "1 year and -25 months" will be normalized to
826     * "-1 year and -1 month".
827     * <p>
828     * This instance is immutable and unaffected by this method call.
829     *
830     * @return a {@code Period} based on this period with excess months normalized to years, not
831     *         null
832     * @throws ArithmeticException if numeric overflow occurs
833     */
834     public Period normalized() {
835         long totalMonths = toTotalMonths();
836         long splitYears = totalMonths / 12;
837         int splitMonths = (int) (totalMonths % 12); // no overflow
838         if (splitYears == years && splitMonths == months) {
839             return this;
840         }
841         return create(Math.toIntExact(splitYears), splitMonths, days);
842     }
843
844     /**
845     * Gets the total number of months in this period.
846     * <p>
847     * This returns the total number of months in the period by multiplying the
848     * number of years by 12 and adding the number of months.
849     * <p>
850     * This instance is immutable and unaffected by this method call.
851     *
852     * @return the total number of months in the period, may be negative
853     */
854     public long toTotalMonths() {
855         return years * 12L + months; // no overflow
856     }
857
858     //-----
859     /**
860     * Adds this period to the specified temporal object.
861     * <p>
862     * This returns a temporal object of the same observable type as the input
863     * with this period added.
864     * If the temporal has a chronology, it must be the ISO chronology.
865     * <p>
866     * In most cases, it is clearer to reverse the calling pattern by using
867     * {@link Temporal#plus(TemporalAmount)}.
868     * <pre>
869     * // these two lines are equivalent, but the second approach is recommended
870     * dateTime = thisPeriod.addTo(dateTime);
871     * dateTime = dateTime.plus(thisPeriod);
872     * </pre>
873     * <p>
874     * The calculation operates as follows.
875     * First, the chronology of the temporal is checked to ensure it is ISO chronology or null.
876     * Second, if the months are zero, the years are added if non-zero, otherwise

```

```

876     * the combination of years and months is added if non-zero.
877     * Finally, any days are added.
878     * <p>
879     * This approach ensures that a partial period can be added to a partial date.
880     * For example, a period of years and/or months can be added to a {@code YearMonth},
881     * but a period including days cannot.
882     * The approach also adds years and months together when necessary, which ensures
883     * correct behaviour at the end of the month.
884     * <p>
885     * This instance is immutable and unaffected by this method call.
886     *
887     * @param temporal the temporal object to adjust, not null
888     * @return an object of the same type with the adjustment made, not null
889     * @throws DateTimeException if unable to add
890     * @throws ArithmeticException if numeric overflow occurs
891     */
892     @Override
893     public Temporal addTo(Temporal temporal) {
894         validateChrono(temporal);
895         if (months == 0) {
896             if (years != 0) {
897                 temporal = temporal.plus(years, YEARS);
898             }
899         } else {
900             long totalMonths = toTotalMonths();
901             if (totalMonths != 0) {
902                 temporal = temporal.plus(totalMonths, MONTHS);
903             }
904         }
905         if (days != 0) {
906             temporal = temporal.plus(days, DAYS);
907         }
908         return temporal;
909     }
910
911     /**
912     * Subtracts this period from the specified temporal object.
913     * <p>
914     * This returns a temporal object of the same observable type as the input
915     * with this period subtracted.
916     * If the temporal has a chronology, it must be the ISO chronology.
917     * <p>
918     * In most cases, it is clearer to reverse the calling pattern by using
919     * {@link Temporal#minus(TemporalAmount)}.
920     * <pre>
921     * // these two lines are equivalent, but the second approach is recommended
922     * dateTime = thisPeriod.subtractFrom(dateTime);
923     * dateTime = dateTime.minus(thisPeriod);
924     * </pre>
925     * <p>
926     * The calculation operates as follows.
927     * First, the chronology of the temporal is checked to ensure it is ISO chronology or null.
928     * Second, if the months are zero, the years are subtracted if non-zero, otherwise

```

```

929     * the combination of years and months is subtracted if non-zero.
930     * Finally, any days are subtracted.
931     * <p>
932     * This approach ensures that a partial period can be subtracted from a partial date.
933     * For example, a period of years and/or months can be subtracted from a {@code YearMonth},
934     * but a period including days cannot.
935     * The approach also subtracts years and months together when necessary, which ensures
936     * correct behaviour at the end of the month.
937     * <p>
938     * This instance is immutable and unaffected by this method call.
939     *
940     * @param temporal the temporal object to adjust, not null
941     * @return an object of the same type with the adjustment made, not null
942     * @throws DateTimeException if unable to subtract
943     * @throws ArithmeticException if numeric overflow occurs
944     */
945     @Override
946     public Temporal subtractFrom(Temporal temporal) {
947         validateChrono(temporal);
948         if (months == 0) {
949             if (years != 0) {
950                 temporal = temporal.minus(years, YEARS);
951             }
952         } else {
953             long totalMonths = toTotalMonths();
954             if (totalMonths != 0) {
955                 temporal = temporal.minus(totalMonths, MONTHS);
956             }
957         }
958         if (days != 0) {
959             temporal = temporal.minus(days, DAYS);
960         }
961         return temporal;
962     }
963
964     /**
965     * Validates that the temporal has the correct chronology.
966     */
967     private void validateChrono(TemporalAccessor temporal) {
968         Objects.requireNonNull(temporal, "temporal");
969         Chronology temporalChrono = temporal.query(TemporalQueries.chronology());
970         if (temporalChrono != null && IsoChronology.INSTANCE.equals(temporalChrono) == false) {
971             throw new DateTimeException("Chronology mismatch, expected: ISO, actual: " +
972                 temporalChrono.getId());
973         }
974     }
975
976     //-----
977     /**
978     * Checks if this period is equal to another period.
979     * <p>
980     * The comparison is based on the type {@code Period} and each of the three amounts.
981     * To be equal, the years, months and days units must be individually equal.

```

```

981     * Note that this means that a period of "15 Months" is not equal to a period
982     * of "1 Year and 3 Months".
983     *
984     * @param obj the object to check, null returns false
985     * @return true if this is equal to the other period
986     */
987     @Override
988     public boolean equals(Object obj) {
989         if (this == obj) {
990             return true;
991         }
992         if (obj instanceof Period) {
993             Period other = (Period) obj;
994             return years == other.years &&
995                 months == other.months &&
996                 days == other.days;
997         }
998         return false;
999     }
1000
1001     /**
1002     * A hash code for this period.
1003     *
1004     * @return a suitable hash code
1005     */
1006     @Override
1007     public int hashCode() {
1008         return years + Integer.rotateLeft(months, 8) + Integer.rotateLeft(days, 16);
1009     }
1010
1011     //-----
1012     /**
1013     * Outputs this period as a {@code String}, such as {@code P6Y3M1D}.
1014     * <p>
1015     * The output will be in the ISO-8601 period format.
1016     * A zero period will be represented as zero days, 'POD'.
1017     *
1018     * @return a string representation of this period, not null
1019     */
1020     @Override
1021     public String toString() {
1022         if (this == ZERO) {
1023             return "POD";
1024         } else {
1025             StringBuilder buf = new StringBuilder();
1026             buf.append('P');
1027             if (years != 0) {
1028                 buf.append(years).append('Y');
1029             }
1030             if (months != 0) {
1031                 buf.append(months).append('M');
1032             }
1033             if (days != 0) {

```

```

1034         buf.append(days).append('D');
1035     }
1036     return buf.toString();
1037 }
1038 }
1039
1040 //-----
1041 /**
1042  * Writes the object using a
1043  * <a href=".../serialized-form.html#java.time.Ser">dedicated serialized form</a>.
1044  * @serialData
1045  * <pre>
1046  *   out.writeByte(14); // identifies a Period
1047  *   out.writeInt(years);
1048  *   out.writeInt(months);
1049  *   out.writeInt(days);
1050  * </pre>
1051  *
1052  * @return the instance of {@code Ser}, not null
1053  */
1054 private Object writeReplace() {
1055     return new Ser(Ser.PERIOD_TYPE, this);
1056 }
1057
1058 /**
1059  * Defend against malicious streams.
1060  *
1061  * @param s the stream to read
1062  * @throws java.io.InvalidObjectException always
1063  */
1064 private void readObject(ObjectInputStream s) throws InvalidObjectException {
1065     throw new InvalidObjectException("Deserialization via serialization delegate");
1066 }
1067
1068 void writeExternal(DataOutput out) throws IOException {
1069     out.writeInt(years);
1070     out.writeInt(months);
1071     out.writeInt(days);
1072 }
1073
1074 static Period readExternal(DataInput in) throws IOException {
1075     int years = in.readInt();
1076     int months = in.readInt();
1077     int days = in.readInt();
1078     return Period.of(years, months, days);
1079 }
1080
1081 }

```

4.14 Ser.java

Secuencia 63: java/time/Ser.java

1 /*

```
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27  * Copyright (c) 2011-2012, Stephen Colebourne & Michael Nascimento Santos
28  *
29  * All rights reserved.
30  *
31  * Redistribution and use in source and binary forms, with or without
32  * modification, are permitted provided that the following conditions are met:
33  *
34  * * Redistributions of source code must retain the above copyright notice,
35  *   this list of conditions and the following disclaimer.
36  *
37  * * Redistributions in binary form must reproduce the above copyright notice,
38  *   this list of conditions and the following disclaimer in the documentation
39  *   and/or other materials provided with the distribution.
40  *
41  * * Neither the name of JSR-310 nor the names of its contributors
42  *   may be used to endorse or promote products derived from this software
43  *   without specific prior written permission.
44  *
45  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
```



```

55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56  */
57  package java.time;
58
59  import java.io.Externalizable;
60  import java.io.IOException;
61  import java.io.InvalidClassException;
62  import java.io.ObjectInput;
63  import java.io.ObjectOutput;
64  import java.io.StreamCorruptedException;
65
66  /**
67   * The shared serialization delegate for this package.
68   *
69   * @implNote
70   * This class wraps the object being serialized, and takes a byte representing the type of the
71   * class to
72   * be serialized. This byte can also be used for versioning the serialization format. In this
73   * case another
74   * byte flag would be used in order to specify an alternative version of the type format.
75   * For example {@code LOCAL_DATE_TYPE_VERSION_2 = 21}.
76   * <p>
77   * In order to serialize the object it writes its byte and then calls back to the appropriate class
78   * where
79   * the serialization is performed. In order to deserialize the object it read in the type byte,
80   * switching
81   * in order to select which class to call back into.
82   * <p>
83   * The serialization format is determined on a per class basis. In the case of field based classes
84   * each
85   * of the fields is written out with an appropriate size format in descending order of the field's
86   * size. For
87   * example in the case of {@link LocalDate} year is written before month. Composite classes, such
88   * as
89   * {@link LocalDateTime} are serialized as one object.
90   * <p>
91   * This class is mutable and should be created once per serialization.
92   *
93   * @serial include
94   * @since 1.8
95   */
96  final class Ser implements Externalizable {
97
98      /**
99       * Serialization version.
100      */
101      private static final long serialVersionUID = -7683839454370182990L;
102
103      static final byte DURATION_TYPE = 1;
104      static final byte INSTANT_TYPE = 2;
105      static final byte LOCAL_DATE_TYPE = 3;
106      static final byte LOCAL_TIME_TYPE = 4;
107      static final byte LOCAL_DATE_TIME_TYPE = 5;

```

```

101 static final byte ZONE_DATE_TIME_TYPE = 6;
102 static final byte ZONE_REGION_TYPE = 7;
103 static final byte ZONE_OFFSET_TYPE = 8;
104 static final byte OFFSET_TIME_TYPE = 9;
105 static final byte OFFSET_DATE_TIME_TYPE = 10;
106 static final byte YEAR_TYPE = 11;
107 static final byte YEAR_MONTH_TYPE = 12;
108 static final byte MONTH_DAY_TYPE = 13;
109 static final byte PERIOD_TYPE = 14;
110
111 /** The type being serialized. */
112 private byte type;
113 /** The object being serialized. */
114 private Object object;
115
116 /**
117  * Constructor for deserialization.
118  */
119 public Ser() {
120 }
121
122 /**
123  * Creates an instance for serialization.
124  *
125  * @param type the type
126  * @param object the object
127  */
128 Ser(byte type, Object object) {
129     this.type = type;
130     this.object = object;
131 }
132
133 //-----
134 /**
135  * Implements the {@code Externalizable} interface to write the object.
136  * @serialData
137  *
138  * Each serializable class is mapped to a type that is the first byte
139  * in the stream. Refer to each class {@code writeReplace}
140  * serialized form for the value of the type and sequence of values for the type.
141  * <ul>
142  * <li><a href="../../serialized-form.html#java.time.Duration">Duration.writeReplace</a>
143  * <li><a href="../../serialized-form.html#java.time.Instant">Instant.writeReplace</a>
144  * <li><a href="../../serialized-form.html#java.time.LocalDate">LocalDate.writeReplace</a>
145  * <li><a href="../../serialized-form.html#java.time.LocalDateTime">LocalDateTime.writeReplace
146  * </a>
147  * <li><a href="../../serialized-form.html#java.time.LocalTime">LocalTime.writeReplace</a>
148  * <li><a href="../../serialized-form.html#java.time.MonthDay">MonthDay.writeReplace</a>
149  * <li><a href="../../serialized-form.html#java.time.OffsetTime">OffsetTime.writeReplace</a>
150  * <li><a href="../../serialized-form.html#java.time.OffsetDateTime">OffsetDateTime.
151  * writeReplace</a>
152  * <li><a href="../../serialized-form.html#java.time.Period">Period.writeReplace</a>
153  * <li><a href="../../serialized-form.html#java.time.Year">Year.writeReplace</a>

```

```

152 * <li><a href="../../../serialized-form.html#java.time.YearMonth">YearMonth.writeReplace</a>
153 * <li><a href="../../../serialized-form.html#java.time.ZoneId">ZoneId.writeReplace</a>
154 * <li><a href="../../../serialized-form.html#java.time.ZoneOffset">ZoneOffset.writeReplace</a>
155 * <li><a href="../../../serialized-form.html#java.time.ZonedDateTime">ZonedDateTime.writeReplace
    </a>
156 * </ul>
157 *
158 * @param out the data stream to write to, not null
159 */
160 @Override
161 public void writeExternal(ObjectOutput out) throws IOException {
162     writeInternal(type, object, out);
163 }
164
165 static void writeInternal(byte type, Object object, ObjectOutput out) throws IOException {
166     out.writeByte(type);
167     switch (type) {
168         case DURATION_TYPE:
169             ((Duration) object).writeExternal(out);
170             break;
171         case INSTANT_TYPE:
172             ((Instant) object).writeExternal(out);
173             break;
174         case LOCAL_DATE_TYPE:
175             ((LocalDate) object).writeExternal(out);
176             break;
177         case LOCAL_DATE_TIME_TYPE:
178             ((LocalDateTime) object).writeExternal(out);
179             break;
180         case LOCAL_TIME_TYPE:
181             ((LocalTime) object).writeExternal(out);
182             break;
183         case ZONE_REGION_TYPE:
184             ((ZoneRegion) object).writeExternal(out);
185             break;
186         case ZONE_OFFSET_TYPE:
187             ((ZoneOffset) object).writeExternal(out);
188             break;
189         case ZONE_DATE_TIME_TYPE:
190             ((ZonedDateTime) object).writeExternal(out);
191             break;
192         case OFFSET_TIME_TYPE:
193             ((OffsetTime) object).writeExternal(out);
194             break;
195         case OFFSET_DATE_TIME_TYPE:
196             ((OffsetDateTime) object).writeExternal(out);
197             break;
198         case YEAR_TYPE:
199             ((Year) object).writeExternal(out);
200             break;
201         case YEAR_MONTH_TYPE:
202             ((YearMonth) object).writeExternal(out);
203             break;

```

```

204         case MONTH_DAY_TYPE:
205             ((MonthDay) object).writeExternal(out);
206             break;
207         case PERIOD_TYPE:
208             ((Period) object).writeExternal(out);
209             break;
210         default:
211             throw new InvalidClassException("Unknown serialized type");
212     }
213 }
214
215 //-----
216 /**
217  * Implements the {@code Externalizable} interface to read the object.
218  * @serialData
219  *
220  * The streamed type and parameters defined by the type's {@code writeReplace}
221  * method are read and passed to the corresponding static factory for the type
222  * to create a new instance. That instance is returned as the de-serialized
223  * {@code Ser} object.
224  *
225  * <ul>
226  * <li><a href="../../serialized-form.html#java.time.Duration">Duration</a> - {@code Duration.
227  *   ofSeconds(seconds, nanos);}
228  * <li><a href="../../serialized-form.html#java.time.Instant">Instant</a> - {@code Instant.
229  *   ofEpochSecond(seconds, nanos);}
230  * <li><a href="../../serialized-form.html#java.time.LocalDate">LocalDate</a> - {@code
231  *   LocalDate.of(year, month, day);}
232  * <li><a href="../../serialized-form.html#java.time.LocalDateTime">LocalDateTime</a> - {@code
233  *   LocalDateTime.of(date, time);}
234  * <li><a href="../../serialized-form.html#java.time.LocalTime">LocalTime</a> - {@code
235  *   LocalTime.of(hour, minute, second, nano);}
236  * <li><a href="../../serialized-form.html#java.time.MonthDay">MonthDay</a> - {@code MonthDay.
237  *   of(month, day);}
238  * <li><a href="../../serialized-form.html#java.time.OffsetTime">OffsetTime</a> - {@code
239  *   OffsetTime.of(time, offset);}
240  * <li><a href="../../serialized-form.html#java.time.OffsetDateTime">OffsetDateTime</a> - {
241  *   @code OffsetDateTime.of(dateTime, offset);}
242  * <li><a href="../../serialized-form.html#java.time.Period">Period</a> - {@code Period.of(
243  *   years, months, days);}
244  * <li><a href="../../serialized-form.html#java.time.Year">Year</a> - {@code Year.of(year);}
245  * <li><a href="../../serialized-form.html#java.time.YearMonth">YearMonth</a> - {@code
246  *   YearMonth.of(year, month);}
247  * <li><a href="../../serialized-form.html#java.time.ZonedDateTime">ZonedDateTime</a> - {@code
248  *   ZonedDateTime.ofLenient(dateTime, offset, zone);}
249  * <li><a href="../../serialized-form.html#java.time.ZoneId">ZoneId</a> - {@code ZoneId.of(id)
250  *   ;}
251  * <li><a href="../../serialized-form.html#java.time.ZoneOffset">ZoneOffset</a> - {@code (
252  *   offsetByte == 127 ? ZoneOffset.ofTotalSeconds(in.readInt()) : ZoneOffset.ofTotalSeconds(
253  *   offsetByte * 900));}
254  * </ul>
255  *
256  * @param in the data to read, not null

```

```

243     */
244     public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
245         type = in.readByte();
246         object = readInternal(type, in);
247     }
248
249     static Object read(ObjectInput in) throws IOException, ClassNotFoundException {
250         byte type = in.readByte();
251         return readInternal(type, in);
252     }
253
254     private static Object readInternal(byte type, ObjectInput in) throws IOException,
        ClassNotFoundException {
255         switch (type) {
256             case DURATION_TYPE: return Duration.readExternal(in);
257             case INSTANT_TYPE: return Instant.readExternal(in);
258             case LOCAL_DATE_TYPE: return LocalDate.readExternal(in);
259             case LOCAL_DATE_TIME_TYPE: return LocalDateTime.readExternal(in);
260             case LOCAL_TIME_TYPE: return LocalTime.readExternal(in);
261             case ZONE_DATE_TIME_TYPE: return ZonedDateTime.readExternal(in);
262             case ZONE_OFFSET_TYPE: return ZoneOffset.readExternal(in);
263             case ZONE_REGION_TYPE: return ZoneRegion.readExternal(in);
264             case OFFSET_TIME_TYPE: return OffsetTime.readExternal(in);
265             case OFFSET_DATE_TIME_TYPE: return OffsetDateTime.readExternal(in);
266             case YEAR_TYPE: return Year.readExternal(in);
267             case YEAR_MONTH_TYPE: return YearMonth.readExternal(in);
268             case MONTH_DAY_TYPE: return MonthDay.readExternal(in);
269             case PERIOD_TYPE: return Period.readExternal(in);
270             default:
271                 throw new StreamCorruptedException("Unknown serialized type");
272         }
273     }
274
275     /**
276     * Returns the object that will replace this one.
277     *
278     * @return the read object, should never be null
279     */
280     private Object readResolve() {
281         return object;
282     }
283
284 }

```

4.15 Year.java

Secuencia 64: java/time/Year.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *

```

```
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
```

```
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time;
63
64  import static java.time.temporal.ChronoField.ERA;
65  import static java.time.temporal.ChronoField.YEAR;
66  import static java.time.temporal.ChronoField.YEAR_OF_ERA;
67  import static java.time.temporal.ChronoUnit.CENTURIES;
68  import static java.time.temporal.ChronoUnit.DECADES;
69  import static java.time.temporal.ChronoUnit.ERAS;
70  import static java.time.temporal.ChronoUnit.MILLENNIA;
71  import static java.time.temporal.ChronoUnit.YEARS;
72
73  import java.io.DataInput;
74  import java.io.DataOutput;
75  import java.io.IOException;
76  import java.io.InvalidObjectException;
77  import java.io.ObjectInputStream;
78  import java.io.Serializable;
79  import java.time.chrono.Chronology;
80  import java.time.chrono.IsoChronology;
81  import java.time.format.DateTimeFormatter;
82  import java.time.format.DateTimeFormatterBuilder;
83  import java.time.format.DateTimeParseException;
84  import java.time.format.SignStyle;
85  import java.time.temporal.ChronoField;
86  import java.time.temporal.ChronoUnit;
87  import java.time.temporal.Temporal;
88  import java.time.temporal.TemporalAccessor;
89  import java.time.temporal.TemporalAdjuster;
90  import java.time.temporal.TemporalAmount;
91  import java.time.temporal.TemporalField;
92  import java.time.temporal.TemporalQueries;
93  import java.time.temporal.TemporalQuery;
94  import java.time.temporal.TemporalUnit;
95  import java.time.temporal.UnsupportedTemporalTypeException;
96  import java.time.temporal.ValueRange;
97  import java.util.Objects;
98
99  /**
100   * A year in the ISO-8601 calendar system, such as {@code 2007}.
101   * <p>
102   * {@code Year} is an immutable date-time object that represents a year.
103   * Any field that can be derived from a year can be obtained.
104   * <p>
105   * <b>Note that years in the ISO chronology only align with years in the
106   * Gregorian-Julian system for modern years. Parts of Russia did not switch to the
107   * modern Gregorian/ISO rules until 1920.
108   * As such, historical years must be treated with caution.</b>
109   * <p>
110   * This class does not store or represent a month, day, time or time-zone.
111   * For example, the value "2007" can be stored in a {@code Year}.
112   * <p>
```

```

113 * Years represented by this class follow the ISO-8601 standard and use
114 * the proleptic numbering system. Year 1 is preceded by year 0, then by year -1.
115 * <p>
116 * The ISO-8601 calendar system is the modern civil calendar system used today
117 * in most of the world. It is equivalent to the proleptic Gregorian calendar
118 * system, in which today's rules for leap years are applied for all time.
119 * For most applications written today, the ISO-8601 rules are entirely suitable.
120 * However, any application that makes use of historical dates, and requires them
121 * to be accurate will find the ISO-8601 approach unsuitable.
122 *
123 * <p>
124 * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
125 * class; use of identity-sensitive operations (including reference equality
126 * ({@code ==}), identity hash code, or synchronization) on instances of
127 * {@code Year} may have unpredictable results and should be avoided.
128 * The {@code equals} method should be used for comparisons.
129 *
130 * @implSpec
131 * This class is immutable and thread-safe.
132 *
133 * @since 1.8
134 */
135 public final class Year
136     implements Temporal, TemporalAdjuster, Comparable<Year>, Serializable {
137
138     /**
139      * The minimum supported year, '-999,999,999'.
140      */
141     public static final int MIN_VALUE = -999_999_999;
142     /**
143      * The maximum supported year, '+999,999,999'.
144      */
145     public static final int MAX_VALUE = 999_999_999;
146
147     /**
148      * Serialization version.
149      */
150     private static final long serialVersionUID = -23038383694477807L;
151     /**
152      * Parser.
153      */
154     private static final DateTimeFormatter PARSER = new DateTimeFormatterBuilder()
155         .appendValue(YEAR, 4, 10, SignStyle.EXCEEDS_PAD)
156         .toFormatter();
157
158     /**
159      * The year being represented.
160      */
161     private final int year;
162
163     //-----
164     /**
165      * Obtains the current year from the system clock in the default time-zone.

```



```

166 * <p>
167 * This will query the {@link Clock#systemDefaultZone() system clock} in the default
168 * time-zone to obtain the current year.
169 * <p>
170 * Using this method will prevent the ability to use an alternate clock for testing
171 * because the clock is hard-coded.
172 *
173 * @return the current year using the system clock and default time-zone, not null
174 */
175 public static Year now() {
176     return now(Clock.systemDefaultZone());
177 }
178
179 /**
180 * Obtains the current year from the system clock in the specified time-zone.
181 * <p>
182 * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current year.
183 * Specifying the time-zone avoids dependence on the default time-zone.
184 * <p>
185 * Using this method will prevent the ability to use an alternate clock for testing
186 * because the clock is hard-coded.
187 *
188 * @param zone the zone ID to use, not null
189 * @return the current year using the system clock, not null
190 */
191 public static Year now(ZoneId zone) {
192     return now(Clock.system(zone));
193 }
194
195 /**
196 * Obtains the current year from the specified clock.
197 * <p>
198 * This will query the specified clock to obtain the current year.
199 * Using this method allows the use of an alternate clock for testing.
200 * The alternate clock may be introduced using {@link Clock dependency injection}.
201 *
202 * @param clock the clock to use, not null
203 * @return the current year, not null
204 */
205 public static Year now(Clock clock) {
206     final LocalDate now = LocalDate.now(clock); // called once
207     return Year.of(now.getYear());
208 }
209
210 //-----
211 /**
212 * Obtains an instance of {@code Year}.
213 * <p>
214 * This method accepts a year value from the proleptic ISO calendar system.
215 * <p>
216 * The year 2AD/CE is represented by 2.<br>
217 * The year 1AD/CE is represented by 1.<br>
218 * The year 1BC/BCE is represented by 0.<br>

```

```

219     * The year 2BC/BCE is represented by -1.<br>
220     *
221     * @param isoYear the ISO proleptic year to represent, from {@code MIN_VALUE} to {@code
      MAX_VALUE}
222     * @return the year, not null
223     * @throws DateTimeException if the field is invalid
224     */
225     public static Year of(int isoYear) {
226         YEAR.checkValidValue(isoYear);
227         return new Year(isoYear);
228     }
229
230     //-----
231     /**
232     * Obtains an instance of {@code Year} from a temporal object.
233     * <p>
234     * This obtains a year based on the specified temporal.
235     * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
236     * which this factory converts to an instance of {@code Year}.
237     * <p>
238     * The conversion extracts the {@link ChronoField#YEAR year} field.
239     * The extraction is only permitted if the temporal object has an ISO
240     * chronology, or can be converted to a {@code LocalDate}.
241     * <p>
242     * This method matches the signature of the functional interface {@link TemporalQuery}
243     * allowing it to be used as a query via method reference, {@code Year::from}.
244     *
245     * @param temporal the temporal object to convert, not null
246     * @return the year, not null
247     * @throws DateTimeException if unable to convert to a {@code Year}
248     */
249     public static Year from(TemporalAccessor temporal) {
250         if (temporal instanceof Year) {
251             return (Year) temporal;
252         }
253         Objects.requireNonNull(temporal, "temporal");
254         try {
255             if (IsoChronology.INSTANCE.equals(Chronology.from(temporal)) == false) {
256                 temporal = LocalDate.from(temporal);
257             }
258             return of(temporal.get(YEAR));
259         } catch (DateTimeException ex) {
260             throw new DateTimeException("Unable to obtain Year from TemporalAccessor: " +
261                 temporal + " of type " + temporal.getClass().getName(), ex);
262         }
263     }
264
265     //-----
266     /**
267     * Obtains an instance of {@code Year} from a text string such as {@code 2007}.
268     * <p>
269     * The string must represent a valid year.
270     * Years outside the range 0000 to 9999 must be prefixed by the plus or minus symbol.

```

```

271     *
272     * @param text  the text to parse such as "2007", not null
273     * @return the parsed year, not null
274     * @throws DateTimeParseException if the text cannot be parsed
275     */
276     public static Year parse(CharSequence text) {
277         return parse(text, PARSE);
278     }
279
280     /**
281     * Obtains an instance of {@code Year} from a text string using a specific formatter.
282     * <p>
283     * The text is parsed using the formatter, returning a year.
284     *
285     * @param text  the text to parse, not null
286     * @param formatter  the formatter to use, not null
287     * @return the parsed year, not null
288     * @throws DateTimeParseException if the text cannot be parsed
289     */
290     public static Year parse(CharSequence text, DateTimeFormatter formatter) {
291         Objects.requireNonNull(formatter, "formatter");
292         return formatter.parse(text, Year::from);
293     }
294
295     //-----
296     /**
297     * Checks if the year is a leap year, according to the ISO proleptic
298     * calendar system rules.
299     * <p>
300     * This method applies the current rules for leap years across the whole time-line.
301     * In general, a year is a leap year if it is divisible by four without
302     * remainder. However, years divisible by 100, are not leap years, with
303     * the exception of years divisible by 400 which are.
304     * <p>
305     * For example, 1904 is a leap year it is divisible by 4.
306     * 1900 was not a leap year as it is divisible by 100, however 2000 was a
307     * leap year as it is divisible by 400.
308     * <p>
309     * The calculation is proleptic - applying the same rules into the far future and far past.
310     * This is historically inaccurate, but is correct for the ISO-8601 standard.
311     *
312     * @param year  the year to check
313     * @return true if the year is leap, false otherwise
314     */
315     public static boolean isLeap(long year) {
316         return ((year & 3) == 0) && ((year % 100) != 0 || (year % 400) == 0);
317     }
318
319     //-----
320     /**
321     * Constructor.
322     *
323     * @param year  the year to represent

```

```

324     */
325     private Year(int year) {
326         this.year = year;
327     }
328
329     //-----
330     /**
331      * Gets the year value.
332      * <p>
333      * The year returned by this method is proleptic as per {@code get(YEAR)}.
334      *
335      * @return the year, {@code MIN_VALUE} to {@code MAX_VALUE}
336      */
337     public int getValue() {
338         return year;
339     }
340
341     //-----
342     /**
343      * Checks if the specified field is supported.
344      * <p>
345      * This checks if this year can be queried for the specified field.
346      * If false, then calling the {@link #range(TemporalField) range},
347      * {@link #get(TemporalField) get} and {@link #with(TemporalField, long)}
348      * methods will throw an exception.
349      * <p>
350      * If the field is a {@link ChronoField} then the query is implemented here.
351      * The supported fields are:
352      * <ul>
353      * <li>{@code YEAR_OF_ERA}
354      * <li>{@code YEAR}
355      * <li>{@code ERA}
356      * </ul>
357      * All other {@code ChronoField} instances will return false.
358      * <p>
359      * If the field is not a {@code ChronoField}, then the result of this method
360      * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
361      * passing {@code this} as the argument.
362      * Whether the field is supported is determined by the field.
363      *
364      * @param field the field to check, null returns false
365      * @return true if the field is supported on this year, false if not
366      */
367     @Override
368     public boolean isSupported(TemporalField field) {
369         if (field instanceof ChronoField) {
370             return field == YEAR || field == YEAR_OF_ERA || field == ERA;
371         }
372         return field != null && field.isSupportedBy(this);
373     }
374
375     /**
376      * Checks if the specified unit is supported.

```

```

377 * <p>
378 * This checks if the specified unit can be added to, or subtracted from, this year.
379 * If false, then calling the {@link #plus(long, TemporalUnit)} and
380 * {@link #minus(long, TemporalUnit) minus} methods will throw an exception.
381 * <p>
382 * If the unit is a {@link ChronoUnit} then the query is implemented here.
383 * The supported units are:
384 * <ul>
385 * <li>{@code YEARS}
386 * <li>{@code DECADES}
387 * <li>{@code CENTURIES}
388 * <li>{@code MILLENNIA}
389 * <li>{@code ERAS}
390 * </ul>
391 * All other {@code ChronoUnit} instances will return false.
392 * <p>
393 * If the unit is not a {@code ChronoUnit}, then the result of this method
394 * is obtained by invoking {@code TemporalUnit.isSupportedBy(Temporal)}
395 * passing {@code this} as the argument.
396 * Whether the unit is supported is determined by the unit.
397 *
398 * @param unit the unit to check, null returns false
399 * @return true if the unit can be added/subtracted, false if not
400 */
401 @Override
402 public boolean isSupported(TemporalUnit unit) {
403     if (unit instanceof ChronoUnit) {
404         return unit == YEARS || unit == DECADES || unit == CENTURIES || unit == MILLENNIA ||
            unit == ERAS;
405     }
406     return unit != null && unit.isSupportedBy(this);
407 }
408
409 //-----
410 /**
411 * Gets the range of valid values for the specified field.
412 * <p>
413 * The range object expresses the minimum and maximum valid values for a field.
414 * This year is used to enhance the accuracy of the returned range.
415 * If it is not possible to return the range, because the field is not supported
416 * or for some other reason, an exception is thrown.
417 * <p>
418 * If the field is a {@link ChronoField} then the query is implemented here.
419 * The {@link #isSupported(TemporalField) supported fields} will return
420 * appropriate range instances.
421 * All other {@code ChronoField} instances will throw an {@code
    UnsupportedTemporalTypeException}.
422 * <p>
423 * If the field is not a {@code ChronoField}, then the result of this method
424 * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
425 * passing {@code this} as the argument.
426 * Whether the range can be obtained is determined by the field.
427 *

```

```

428     * @param field the field to query the range for, not null
429     * @return the range of valid values for the field, not null
430     * @throws DateTimeException if the range for the field cannot be obtained
431     * @throws UnsupportedTemporalTypeException if the field is not supported
432     */
433     @Override
434     public ValueRange range(TemporalField field) {
435         if (field == YEAR_OF_ERA) {
436             return (year <= 0 ? ValueRange.of(1, MAX_VALUE + 1) : ValueRange.of(1, MAX_VALUE));
437         }
438         return Temporal.super.range(field);
439     }
440
441     /**
442     * Gets the value of the specified field from this year as an {@code int}.
443     * <p>
444     * This queries this year for the value of the specified field.
445     * The returned value will always be within the valid range of values for the field.
446     * If it is not possible to return the value, because the field is not supported
447     * or for some other reason, an exception is thrown.
448     * <p>
449     * If the field is a {@link ChronoField} then the query is implemented here.
450     * The {@link #isSupported(TemporalField) supported fields} will return valid
451     * values based on this year.
452     * All other {@code ChronoField} instances will throw an {@code
453         UnsupportedTemporalTypeException}.
454     * <p>
455     * If the field is not a {@code ChronoField}, then the result of this method
456     * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
457     * passing {@code this} as the argument. Whether the value can be obtained,
458     * and what the value represents, is determined by the field.
459     *
460     * @param field the field to get, not null
461     * @return the value for the field
462     * @throws DateTimeException if a value for the field cannot be obtained or
463     *         the value is outside the range of valid values for the field
464     * @throws UnsupportedTemporalTypeException if the field is not supported or
465     *         the range of values exceeds an {@code int}
466     * @throws ArithmeticException if numeric overflow occurs
467     */
468     @Override // override for Javadoc
469     public int get(TemporalField field) {
470         return range(field).checkValidIntValue(getLong(field), field);
471     }
472
473     /**
474     * Gets the value of the specified field from this year as a {@code long}.
475     * <p>
476     * This queries this year for the value of the specified field.
477     * If it is not possible to return the value, because the field is not supported
478     * or for some other reason, an exception is thrown.
479     * <p>
480     * If the field is a {@link ChronoField} then the query is implemented here.

```

```

480 * The {@link #isSupported(TemporalField) supported fields} will return valid
481 * values based on this year.
482 * All other {@code ChronoField} instances will throw an {@code
    UnsupportedTemporalTypeException}.
483 * <p>
484 * If the field is not a {@code ChronoField}, then the result of this method
485 * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
486 * passing {@code this} as the argument. Whether the value can be obtained,
487 * and what the value represents, is determined by the field.
488 *
489 * @param field the field to get, not null
490 * @return the value for the field
491 * @throws DateTimeException if a value for the field cannot be obtained
492 * @throws UnsupportedTemporalTypeException if the field is not supported
493 * @throws ArithmeticException if numeric overflow occurs
494 */
495 @Override
496 public long getLong(TemporalField field) {
497     if (field instanceof ChronoField) {
498         switch ((ChronoField) field) {
499             case YEAR_OF_ERA: return (year < 1 ? 1 - year : year);
500             case YEAR: return year;
501             case ERA: return (year < 1 ? 0 : 1);
502         }
503         throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
504     }
505     return field.getFrom(this);
506 }
507
508 //-----
509 /**
510 * Checks if the year is a leap year, according to the ISO proleptic
511 * calendar system rules.
512 * <p>
513 * This method applies the current rules for leap years across the whole time-line.
514 * In general, a year is a leap year if it is divisible by four without
515 * remainder. However, years divisible by 100, are not leap years, with
516 * the exception of years divisible by 400 which are.
517 * <p>
518 * For example, 1904 is a leap year it is divisible by 4.
519 * 1900 was not a leap year as it is divisible by 100, however 2000 was a
520 * leap year as it is divisible by 400.
521 * <p>
522 * The calculation is proleptic - applying the same rules into the far future and far past.
523 * This is historically inaccurate, but is correct for the ISO-8601 standard.
524 *
525 * @return true if the year is leap, false otherwise
526 */
527 public boolean isLeap() {
528     return Year.isLeap(year);
529 }
530
531 /**

```

```

532     * Checks if the month-day is valid for this year.
533     * <p>
534     * This method checks whether this year and the input month and day form
535     * a valid date.
536     *
537     * @param monthDay the month-day to validate, null returns false
538     * @return true if the month and day are valid for this year
539     */
540     public boolean isValidMonthDay(MonthDay monthDay) {
541         return monthDay != null && monthDay.isValidYear(year);
542     }
543
544     /**
545     * Gets the length of this year in days.
546     *
547     * @return the length of this year in days, 365 or 366
548     */
549     public int length() {
550         return isLeap() ? 366 : 365;
551     }
552
553     //-----
554     /**
555     * Returns an adjusted copy of this year.
556     * <p>
557     * This returns a {@code Year}, based on this one, with the year adjusted.
558     * The adjustment takes place using the specified adjuster strategy object.
559     * Read the documentation of the adjuster to understand what adjustment will be made.
560     * <p>
561     * The result of this method is obtained by invoking the
562     * {@link TemporalAdjuster#adjustInto(Temporal)} method on the
563     * specified adjuster passing {@code this} as the argument.
564     * <p>
565     * This instance is immutable and unaffected by this method call.
566     *
567     * @param adjuster the adjuster to use, not null
568     * @return a {@code Year} based on {@code this} with the adjustment made, not null
569     * @throws DateTimeException if the adjustment cannot be made
570     * @throws ArithmeticException if numeric overflow occurs
571     */
572     @Override
573     public Year with(TemporalAdjuster adjuster) {
574         return (Year) adjuster.adjustInto(this);
575     }
576
577     /**
578     * Returns a copy of this year with the specified field set to a new value.
579     * <p>
580     * This returns a {@code Year}, based on this one, with the value
581     * for the specified field changed.
582     * If it is not possible to set the value, because the field is not supported or for
583     * some other reason, an exception is thrown.
584     * <p>

```



```

585     * If the field is a {@link ChronoField} then the adjustment is implemented here.
586     * The supported fields behave as follows:
587     * <ul>
588     * <li>{@code YEAR_OF_ERA} -
589     * Returns a {@code Year} with the specified year-of-era
590     * The era will be unchanged.
591     * <li>{@code YEAR} -
592     * Returns a {@code Year} with the specified year.
593     * This completely replaces the date and is equivalent to {@link #of(int)}.
594     * <li>{@code ERA} -
595     * Returns a {@code Year} with the specified era.
596     * The year-of-era will be unchanged.
597     * </ul>
598     * <p>
599     * In all cases, if the new value is outside the valid range of values for the field
600     * then a {@code DateTimeException} will be thrown.
601     * <p>
602     * All other {@code ChronoField} instances will throw an {@code
        UnsupportedTemporalTypeException}.
603     * <p>
604     * If the field is not a {@code ChronoField}, then the result of this method
605     * is obtained by invoking {@code TemporalField.adjustInto(Temporal, long)}
606     * passing {@code this} as the argument. In this case, the field determines
607     * whether and how to adjust the instant.
608     * <p>
609     * This instance is immutable and unaffected by this method call.
610     *
611     * @param field the field to set in the result, not null
612     * @param newValue the new value of the field in the result
613     * @return a {@code Year} based on {@code this} with the specified field set, not null
614     * @throws DateTimeException if the field cannot be set
615     * @throws UnsupportedTemporalTypeException if the field is not supported
616     * @throws ArithmeticException if numeric overflow occurs
617     */
618     @Override
619     public Year with(TemporalField field, long newValue) {
620         if (field instanceof ChronoField) {
621             ChronoField f = (ChronoField) field;
622             f.checkValidValue(newValue);
623             switch (f) {
624                 case YEAR_OF_ERA: return Year.of((int) (year < 1 ? 1 - newValue : newValue));
625                 case YEAR: return Year.of((int) newValue);
626                 case ERA: return (getLong(ERA) == newValue ? this : Year.of(1 - year));
627             }
628             throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
629         }
630         return field.adjustInto(this, newValue);
631     }
632
633     //-----
634     /**
635     * Returns a copy of this year with the specified amount added.
636     * <p>

```

```

637 * This returns a {@code Year}, based on this one, with the specified amount added.
638 * The amount is typically {@link Period} but may be any other type implementing
639 * the {@link TemporalAmount} interface.
640 * <p>
641 * The calculation is delegated to the amount object by calling
642 * {@link TemporalAmount#addTo(Temporal)}. The amount implementation is free
643 * to implement the addition in any way it wishes, however it typically
644 * calls back to {@link #plus(long, TemporalUnit)}. Consult the documentation
645 * of the amount implementation to determine if it can be successfully added.
646 * <p>
647 * This instance is immutable and unaffected by this method call.
648 *
649 * @param amountToAdd the amount to add, not null
650 * @return a {@code Year} based on this year with the addition made, not null
651 * @throws DateTimeException if the addition cannot be made
652 * @throws ArithmeticException if numeric overflow occurs
653 */
654 @Override
655 public Year plus(TemporalAmount amountToAdd) {
656     return (Year) amountToAdd.addTo(this);
657 }
658
659 /**
660 * Returns a copy of this year with the specified amount added.
661 * <p>
662 * This returns a {@code Year}, based on this one, with the amount
663 * in terms of the unit added. If it is not possible to add the amount, because the
664 * unit is not supported or for some other reason, an exception is thrown.
665 * <p>
666 * If the field is a {@link ChronoUnit} then the addition is implemented here.
667 * The supported fields behave as follows:
668 * <ul>
669 * <li>{@code YEARS} -
670 * Returns a {@code Year} with the specified number of years added.
671 * This is equivalent to {@link #plusYears(long)}.
672 * <li>{@code DECADES} -
673 * Returns a {@code Year} with the specified number of decades added.
674 * This is equivalent to calling {@link #plusYears(long)} with the amount
675 * multiplied by 10.
676 * <li>{@code CENTURIES} -
677 * Returns a {@code Year} with the specified number of centuries added.
678 * This is equivalent to calling {@link #plusYears(long)} with the amount
679 * multiplied by 100.
680 * <li>{@code MILLENNIA} -
681 * Returns a {@code Year} with the specified number of millennia added.
682 * This is equivalent to calling {@link #plusYears(long)} with the amount
683 * multiplied by 1,000.
684 * <li>{@code ERAS} -
685 * Returns a {@code Year} with the specified number of eras added.
686 * Only two eras are supported so the amount must be one, zero or minus one.
687 * If the amount is non-zero then the year is changed such that the year-of-era
688 * is unchanged.
689 * </ul>

```

```

690 * <p>
691 * All other {@code ChronoUnit} instances will throw an {@code UnsupportedOperationException}
692 *   }.
693 * <p>
694 * If the field is not a {@code ChronoUnit}, then the result of this method
695 * is obtained by invoking {@code TemporalUnit.addTo(Temporal, long)}
696 * passing {@code this} as the argument. In this case, the unit determines
697 * whether and how to perform the addition.
698 * <p>
699 * This instance is immutable and unaffected by this method call.
700 *
701 * @param amountToAdd the amount of the unit to add to the result, may be negative
702 * @param unit the unit of the amount to add, not null
703 * @return a {@code Year} based on this year with the specified amount added, not null
704 * @throws DateTimeException if the addition cannot be made
705 * @throws UnsupportedOperationException if the unit is not supported
706 * @throws ArithmeticException if numeric overflow occurs
707 */
708 @Override
709 public Year plus(long amountToAdd, TemporalUnit unit) {
710     if (unit instanceof ChronoUnit) {
711         switch ((ChronoUnit) unit) {
712             case YEARS: return plusYears(amountToAdd);
713             case DECADES: return plusYears(Math.multiplyExact(amountToAdd, 10));
714             case CENTURIES: return plusYears(Math.multiplyExact(amountToAdd, 100));
715             case MILLENNIA: return plusYears(Math.multiplyExact(amountToAdd, 1000));
716             case ERAS: return with(ERA, Math.addExact(getLong(ERA), amountToAdd));
717         }
718         throw new UnsupportedOperationException("Unsupported unit: " + unit);
719     }
720     return unit.addTo(this, amountToAdd);
721 }
722 /**
723 * Returns a copy of this {@code Year} with the specified number of years added.
724 * <p>
725 * This instance is immutable and unaffected by this method call.
726 *
727 * @param yearsToAdd the years to add, may be negative
728 * @return a {@code Year} based on this year with the years added, not null
729 * @throws DateTimeException if the result exceeds the supported range
730 */
731 public Year plusYears(long yearsToAdd) {
732     if (yearsToAdd == 0) {
733         return this;
734     }
735     return of(YEAR.checkValidIntValue(year + yearsToAdd)); // overflow safe
736 }
737
738 //-----
739 /**
740 * Returns a copy of this year with the specified amount subtracted.
741 * <p>

```

```

742 * This returns a {@code Year}, based on this one, with the specified amount subtracted.
743 * The amount is typically {@link Period} but may be any other type implementing
744 * the {@link TemporalAmount} interface.
745 * <p>
746 * The calculation is delegated to the amount object by calling
747 * {@link TemporalAmount#subtractFrom(Temporal)}. The amount implementation is free
748 * to implement the subtraction in any way it wishes, however it typically
749 * calls back to {@link #minus(long, TemporalUnit)}. Consult the documentation
750 * of the amount implementation to determine if it can be successfully subtracted.
751 * <p>
752 * This instance is immutable and unaffected by this method call.
753 *
754 * @param amountToSubtract the amount to subtract, not null
755 * @return a {@code Year} based on this year with the subtraction made, not null
756 * @throws DateTimeException if the subtraction cannot be made
757 * @throws ArithmeticException if numeric overflow occurs
758 */
759 @Override
760 public Year minus(TemporalAmount amountToSubtract) {
761     return (Year) amountToSubtract.subtractFrom(this);
762 }
763
764 /**
765 * Returns a copy of this year with the specified amount subtracted.
766 * <p>
767 * This returns a {@code Year}, based on this one, with the amount
768 * in terms of the unit subtracted. If it is not possible to subtract the amount,
769 * because the unit is not supported or for some other reason, an exception is thrown.
770 * <p>
771 * This method is equivalent to {@link #plus(long, TemporalUnit)} with the amount negated.
772 * See that method for a full description of how addition, and thus subtraction, works.
773 * <p>
774 * This instance is immutable and unaffected by this method call.
775 *
776 * @param amountToSubtract the amount of the unit to subtract from the result, may be negative
777 * @param unit the unit of the amount to subtract, not null
778 * @return a {@code Year} based on this year with the specified amount subtracted, not null
779 * @throws DateTimeException if the subtraction cannot be made
780 * @throws UnsupportedTemporalTypeException if the unit is not supported
781 * @throws ArithmeticException if numeric overflow occurs
782 */
783 @Override
784 public Year minus(long amountToSubtract, TemporalUnit unit) {
785     return (amountToSubtract == Long.MIN_VALUE ? plus(Long.MAX_VALUE, unit).plus(1, unit) :
786         plus(-amountToSubtract, unit));
787 }
788
789 /**
790 * Returns a copy of this {@code Year} with the specified number of years subtracted.
791 * <p>
792 * This instance is immutable and unaffected by this method call.
793 *
794 * @param yearsToSubtract the years to subtract, may be negative

```

```

794     * @return a {@code Year} based on this year with the year subtracted, not null
795     * @throws DateTimeException if the result exceeds the supported range
796     */
797     public Year minusYears(long yearsToSubtract) {
798         return (yearsToSubtract == Long.MIN_VALUE ? plusYears(Long.MAX_VALUE).plusYears(1) :
799             plusYears(-yearsToSubtract));
800     }
801
802     /**
803     * Queries this year using the specified query.
804     * <p>
805     * This queries this year using the specified query strategy object.
806     * The {@code TemporalQuery} object defines the logic to be used to
807     * obtain the result. Read the documentation of the query to understand
808     * what the result of this method will be.
809     * <p>
810     * The result of this method is obtained by invoking the
811     * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
812     * specified query passing {@code this} as the argument.
813     *
814     * @param <R> the type of the result
815     * @param query the query to invoke, not null
816     * @return the query result, null may be returned (defined by the query)
817     * @throws DateTimeException if unable to query (defined by the query)
818     * @throws ArithmeticException if numeric overflow occurs (defined by the query)
819     */
820     @SuppressWarnings("unchecked")
821     @Override
822     public <R> query(TemporalQuery<R> query) {
823         if (query == TemporalQueries.chronology()) {
824             return (R) IsoChronology.INSTANCE;
825         } else if (query == TemporalQueries.precision()) {
826             return (R) YEARS;
827         }
828         return Temporal.super.query(query);
829     }
830
831     /**
832     * Adjusts the specified temporal object to have this year.
833     * <p>
834     * This returns a temporal object of the same observable type as the input
835     * with the year changed to be the same as this.
836     * <p>
837     * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
838     * passing {@link ChronoField#YEAR} as the field.
839     * If the specified temporal object does not use the ISO calendar system then
840     * a {@code DateTimeException} is thrown.
841     * <p>
842     * In most cases, it is clearer to reverse the calling pattern by using
843     * {@link Temporal#with(TemporalAdjuster)}:
844     * <pre>
845     * // these two lines are equivalent, but the second approach is recommended

```

```

846     * temporal = thisYear.adjustInto(temporal);
847     * temporal = temporal.with(thisYear);
848     * </pre>
849     * <p>
850     * This instance is immutable and unaffected by this method call.
851     *
852     * @param temporal the target object to be adjusted, not null
853     * @return the adjusted object, not null
854     * @throws DateTimeException if unable to make the adjustment
855     * @throws ArithmeticException if numeric overflow occurs
856     */
857     @Override
858     public Temporal adjustInto(Temporal temporal) {
859         if (Chronology.from(temporal).equals(IsoChronology.INSTANCE) == false) {
860             throw new DateTimeException("Adjustment only supported on ISO date-time");
861         }
862         return temporal.with(YEAR, year);
863     }
864
865     /**
866     * Calculates the amount of time until another year in terms of the specified unit.
867     * <p>
868     * This calculates the amount of time between two {@code Year}
869     * objects in terms of a single {@code TemporalUnit}.
870     * The start and end points are {@code this} and the specified year.
871     * The result will be negative if the end is before the start.
872     * The {@code Temporal} passed to this method is converted to a
873     * {@code Year} using {@link #from(TemporalAccessor)}.
874     * For example, the amount in decades between two year can be calculated
875     * using {@code startYear.until(endYear, DECADES)}.
876     * <p>
877     * The calculation returns a whole number, representing the number of
878     * complete units between the two years.
879     * For example, the amount in decades between 2012 and 2031
880     * will only be one decade as it is one year short of two decades.
881     * <p>
882     * There are two equivalent ways of using this method.
883     * The first is to invoke this method.
884     * The second is to use {@link TemporalUnit#between(Temporal, Temporal)}:
885     * <pre>
886     * // these two lines are equivalent
887     * amount = start.until(end, YEARS);
888     * amount = YEARS.between(start, end);
889     * </pre>
890     * The choice should be made based on which makes the code more readable.
891     * <p>
892     * The calculation is implemented in this method for {@link ChronoUnit}.
893     * The units {@code YEARS}, {@code DECADES}, {@code CENTURIES},
894     * {@code MILLENNIA} and {@code ERAS} are supported.
895     * Other {@code ChronoUnit} values will throw an exception.
896     * <p>
897     * If the unit is not a {@code ChronoUnit}, then the result of this method
898     * is obtained by invoking {@code TemporalUnit.between(Temporal, Temporal)}

```

```

899     * passing {@code this} as the first argument and the converted input temporal
900     * as the second argument.
901     * <p>
902     * This instance is immutable and unaffected by this method call.
903     *
904     * @param endExclusive the end date, exclusive, which is converted to a {@code Year}, not null
905     * @param unit the unit to measure the amount in, not null
906     * @return the amount of time between this year and the end year
907     * @throws DateTimeException if the amount cannot be calculated, or the end
908     *         temporal cannot be converted to a {@code Year}
909     * @throws UnsupportedTemporalTypeException if the unit is not supported
910     * @throws ArithmeticException if numeric overflow occurs
911     */
912     @Override
913     public long until(Temporal endExclusive, TemporalUnit unit) {
914         Year end = Year.from(endExclusive);
915         if (unit instanceof ChronoUnit) {
916             long yearsUntil = ((long) end.year) - year; // no overflow
917             switch ((ChronoUnit) unit) {
918                 case YEARS: return yearsUntil;
919                 case DECADES: return yearsUntil / 10;
920                 case CENTURIES: return yearsUntil / 100;
921                 case MILLENNIA: return yearsUntil / 1000;
922                 case ERAS: return end.getLong(ERA) - getLong(ERA);
923             }
924             throw new UnsupportedTemporalTypeException("Unsupported unit: " + unit);
925         }
926         return unit.between(this, end);
927     }
928
929     /**
930     * Formats this year using the specified formatter.
931     * <p>
932     * This year will be passed to the formatter to produce a string.
933     *
934     * @param formatter the formatter to use, not null
935     * @return the formatted year string, not null
936     * @throws DateTimeException if an error occurs during printing
937     */
938     public String format(DateTimeFormatter formatter) {
939         Objects.requireNonNull(formatter, "formatter");
940         return formatter.format(this);
941     }
942
943     //-----
944     /**
945     * Combines this year with a day-of-year to create a {@code LocalDate}.
946     * <p>
947     * This returns a {@code LocalDate} formed from this year and the specified day-of-year.
948     * <p>
949     * The day-of-year value 366 is only valid in a leap year.
950     *
951     * @param dayOfYear the day-of-year to use, from 1 to 365-366

```



```
952     * @return the local date formed from this year and the specified date of year, not null
953     * @throws DateTimeException if the day of year is zero or less, 366 or greater or equal
954     * to 366 and this is not a leap year
955     */
956     public LocalDate atDay(int dayOfYear) {
957         return LocalDate.ofYearDay(year, dayOfYear);
958     }
959
960     /**
961     * Combines this year with a month to create a {@code YearMonth}.
962     * <p>
963     * This returns a {@code YearMonth} formed from this year and the specified month.
964     * All possible combinations of year and month are valid.
965     * <p>
966     * This method can be used as part of a chain to produce a date:
967     * <pre>
968     *   LocalDate date = year.atMonth(month).atDay(day);
969     * </pre>
970     *
971     * @param month the month-of-year to use, not null
972     * @return the year-month formed from this year and the specified month, not null
973     */
974     public YearMonth atMonth(Month month) {
975         return YearMonth.of(year, month);
976     }
977
978     /**
979     * Combines this year with a month to create a {@code YearMonth}.
980     * <p>
981     * This returns a {@code YearMonth} formed from this year and the specified month.
982     * All possible combinations of year and month are valid.
983     * <p>
984     * This method can be used as part of a chain to produce a date:
985     * <pre>
986     *   LocalDate date = year.atMonth(month).atDay(day);
987     * </pre>
988     *
989     * @param month the month-of-year to use, from 1 (January) to 12 (December)
990     * @return the year-month formed from this year and the specified month, not null
991     * @throws DateTimeException if the month is invalid
992     */
993     public YearMonth atMonth(int month) {
994         return YearMonth.of(year, month);
995     }
996
997     /**
998     * Combines this year with a month-day to create a {@code LocalDate}.
999     * <p>
1000    * This returns a {@code LocalDate} formed from this year and the specified month-day.
1001    * <p>
1002    * A month-day of February 29th will be adjusted to February 28th in the resulting
1003    * date if the year is not a leap year.
1004    *
```



```

1005     * @param monthDay the month-day to use, not null
1006     * @return the local date formed from this year and the specified month-day, not null
1007     */
1008     public LocalDate atMonthDay(MonthDay monthDay) {
1009         return monthDay.atYear(year);
1010     }
1011
1012     //-----
1013     /**
1014      * Compares this year to another year.
1015      * <p>
1016      * The comparison is based on the value of the year.
1017      * It is "consistent with equals", as defined by {@link Comparable}.
1018      *
1019      * @param other the other year to compare to, not null
1020      * @return the comparator value, negative if less, positive if greater
1021      */
1022     @Override
1023     public int compareTo(Year other) {
1024         return year - other.year;
1025     }
1026
1027     /**
1028      * Checks if this year is after the specified year.
1029      *
1030      * @param other the other year to compare to, not null
1031      * @return true if this is after the specified year
1032      */
1033     public boolean isAfter(Year other) {
1034         return year > other.year;
1035     }
1036
1037     /**
1038      * Checks if this year is before the specified year.
1039      *
1040      * @param other the other year to compare to, not null
1041      * @return true if this point is before the specified year
1042      */
1043     public boolean isBefore(Year other) {
1044         return year < other.year;
1045     }
1046
1047     //-----
1048     /**
1049      * Checks if this year is equal to another year.
1050      * <p>
1051      * The comparison is based on the time-line position of the years.
1052      *
1053      * @param obj the object to check, null returns false
1054      * @return true if this is equal to the other year
1055      */
1056     @Override
1057     public boolean equals(Object obj) {

```

```

1058     if (this == obj) {
1059         return true;
1060     }
1061     if (obj instanceof Year) {
1062         return year == ((Year) obj).year;
1063     }
1064     return false;
1065 }
1066
1067 /**
1068  * A hash code for this year.
1069  *
1070  * @return a suitable hash code
1071  */
1072 @Override
1073 public int hashCode() {
1074     return year;
1075 }
1076
1077 //-----
1078 /**
1079  * Outputs this year as a {@code String}.
1080  *
1081  * @return a string representation of this year, not null
1082  */
1083 @Override
1084 public String toString() {
1085     return Integer.toString(year);
1086 }
1087
1088 //-----
1089 /**
1090  * Writes the object using a
1091  * <a href="../../serialized-form.html#java.time.Ser">dedicated serialized form</a>.
1092  * @serialData
1093  * <pre>
1094  *   out.writeByte(11); // identifies a Year
1095  *   out.writeInt(year);
1096  * </pre>
1097  *
1098  * @return the instance of {@code Ser}, not null
1099  */
1100 private Object writeReplace() {
1101     return new Ser(Ser.YEAR_TYPE, this);
1102 }
1103
1104 /**
1105  * Defend against malicious streams.
1106  *
1107  * @param s the stream to read
1108  * @throws InvalidObjectException always
1109  */
1110 private void readObject(ObjectInputStream s) throws InvalidObjectException {

```

```
1111     throw new InvalidObjectException("Deserialization via serialization delegate");
1112 }
1113
1114 void writeExternal(DataOutput out) throws IOException {
1115     out.writeInt(year);
1116 }
1117
1118 static Year readExternal(DataInput in) throws IOException {
1119     return Year.of(in.readInt());
1120 }
1121
1122 }
```

4.16 YearMonth.java

Secuencia 65: java/time/YearMonth.java

```
1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
```

```
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62 package java.time;
63
64 import static java.time.temporal.ChronoField.ERA;
65 import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
66 import static java.time.temporal.ChronoField.PROLEPTIC_MONTH;
67 import static java.time.temporal.ChronoField.YEAR;
68 import static java.time.temporal.ChronoField.YEAR_OF_ERA;
69 import static java.time.temporal.ChronoUnit.CENTURIES;
70 import static java.time.temporal.ChronoUnit.DECADES;
71 import static java.time.temporal.ChronoUnit.ERAS;
72 import static java.time.temporal.ChronoUnit.MILLENNIA;
73 import static java.time.temporal.ChronoUnit.MONTHS;
74 import static java.time.temporal.ChronoUnit.YEARS;
75
76 import java.io.DataInput;
77 import java.io.DataOutput;
78 import java.io.IOException;
79 import java.io.InvalidObjectException;
80 import java.io.ObjectInputStream;
81 import java.io.Serializable;
82 import java.time.chrono.Chronology;
83 import java.time.chrono.IsoChronology;
84 import java.time.format.DateTimeFormatter;
85 import java.time.format.DateTimeFormatterBuilder;
86 import java.time.format.DateTimeParseException;
87 import java.time.format.SignStyle;
88 import java.time.temporal.ChronoField;
89 import java.time.temporal.ChronoUnit;
90 import java.time.temporal.Temporal;
```

```

91 import java.time.temporal.TemporalAccessor;
92 import java.time.temporal.TemporalAdjuster;
93 import java.time.temporal.TemporalAmount;
94 import java.time.temporal.TemporalField;
95 import java.time.temporal.TemporalQueries;
96 import java.time.temporal.TemporalQuery;
97 import java.time.temporal.TemporalUnit;
98 import java.time.temporal.UnsupportedTemporalTypeException;
99 import java.time.temporal.ValueRange;
100 import java.util.Objects;
101
102 /**
103  * A year-month in the ISO-8601 calendar system, such as {@code 2007-12}.
104  * <p>
105  * {@code YearMonth} is an immutable date-time object that represents the combination
106  * of a year and month. Any field that can be derived from a year and month, such as
107  * quarter-of-year, can be obtained.
108  * <p>
109  * This class does not store or represent a day, time or time-zone.
110  * For example, the value "October 2007" can be stored in a {@code YearMonth}.
111  * <p>
112  * The ISO-8601 calendar system is the modern civil calendar system used today
113  * in most of the world. It is equivalent to the proleptic Gregorian calendar
114  * system, in which today's rules for leap years are applied for all time.
115  * For most applications written today, the ISO-8601 rules are entirely suitable.
116  * However, any application that makes use of historical dates, and requires them
117  * to be accurate will find the ISO-8601 approach unsuitable.
118  *
119  * <p>
120  * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
121  * class; use of identity-sensitive operations (including reference equality
122  * ({@code ==}), identity hash code, or synchronization) on instances of
123  * {@code YearMonth} may have unpredictable results and should be avoided.
124  * The {@code equals} method should be used for comparisons.
125  *
126  * @implSpec
127  * This class is immutable and thread-safe.
128  *
129  * @since 1.8
130  */
131 public final class YearMonth
132     implements Temporal, TemporalAdjuster, Comparable<YearMonth>, Serializable {
133
134     /**
135      * Serialization version.
136      */
137     private static final long serialVersionUID = 4183400860270640070L;
138
139     /**
140      * Parser.
141      */
142     private static final DateTimeFormatter PARSER = new DateTimeFormatterBuilder()
143         .appendValue(YEAR, 4, 10, SignStyle.EXCEEDS_PAD)
144         .appendLiteral('-')

```

```

144     .appendValue(MONTH_OF_YEAR, 2)
145     .toFormatter();
146
147     /**
148      * The year.
149      */
150     private final int year;
151     /**
152      * The month-of-year, not null.
153      */
154     private final int month;
155
156     //-----
157     /**
158      * Obtains the current year-month from the system clock in the default time-zone.
159      * <p>
160      * This will query the {@link Clock#systemDefaultZone() system clock} in the default
161      * time-zone to obtain the current year-month.
162      * <p>
163      * Using this method will prevent the ability to use an alternate clock for testing
164      * because the clock is hard-coded.
165      *
166      * @return the current year-month using the system clock and default time-zone, not null
167      */
168     public static YearMonth now() {
169         return now(Clock.systemDefaultZone());
170     }
171
172     /**
173      * Obtains the current year-month from the system clock in the specified time-zone.
174      * <p>
175      * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current year-
176      * month.
177      * Specifying the time-zone avoids dependence on the default time-zone.
178      * <p>
179      * Using this method will prevent the ability to use an alternate clock for testing
180      * because the clock is hard-coded.
181      *
182      * @param zone the zone ID to use, not null
183      * @return the current year-month using the system clock, not null
184      */
185     public static YearMonth now(ZoneId zone) {
186         return now(Clock.system(zone));
187     }
188
189     /**
190      * Obtains the current year-month from the specified clock.
191      * <p>
192      * This will query the specified clock to obtain the current year-month.
193      * Using this method allows the use of an alternate clock for testing.
194      * The alternate clock may be introduced using {@link Clock dependency injection}.
195      *
196      * @param clock the clock to use, not null

```

```

196     * @return the current year-month, not null
197     */
198     public static YearMonth now(Clock clock) {
199         final LocalDate now = LocalDate.now(clock); // called once
200         return YearMonth.of(now.getYear(), now.getMonth());
201     }
202
203     //-----
204     /**
205      * Obtains an instance of {@code YearMonth} from a year and month.
206      *
207      * @param year the year to represent, from MIN_YEAR to MAX_YEAR
208      * @param month the month-of-year to represent, not null
209      * @return the year-month, not null
210      * @throws DateTimeException if the year value is invalid
211      */
212     public static YearMonth of(int year, Month month) {
213         Objects.requireNonNull(month, "month");
214         return of(year, month.getValue());
215     }
216
217     /**
218      * Obtains an instance of {@code YearMonth} from a year and month.
219      *
220      * @param year the year to represent, from MIN_YEAR to MAX_YEAR
221      * @param month the month-of-year to represent, from 1 (January) to 12 (December)
222      * @return the year-month, not null
223      * @throws DateTimeException if either field value is invalid
224      */
225     public static YearMonth of(int year, int month) {
226         YEAR.checkValidValue(year);
227         MONTH_OF_YEAR.checkValidValue(month);
228         return new YearMonth(year, month);
229     }
230
231     //-----
232     /**
233      * Obtains an instance of {@code YearMonth} from a temporal object.
234      * <p>
235      * This obtains a year-month based on the specified temporal.
236      * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
237      * which this factory converts to an instance of {@code YearMonth}.
238      * <p>
239      * The conversion extracts the {@link ChronoField#YEAR YEAR} and
240      * {@link ChronoField#MONTH_OF_YEAR MONTH_OF_YEAR} fields.
241      * The extraction is only permitted if the temporal object has an ISO
242      * chronology, or can be converted to a {@code LocalDate}.
243      * <p>
244      * This method matches the signature of the functional interface {@link TemporalQuery}
245      * allowing it to be used as a query via method reference, {@code YearMonth::from}.
246      *
247      * @param temporal the temporal object to convert, not null
248      * @return the year-month, not null

```

```

249     * @throws DateTimeException if unable to convert to a {@code YearMonth}
250     */
251     public static YearMonth from(TemporalAccessor temporal) {
252         if (temporal instanceof YearMonth) {
253             return (YearMonth) temporal;
254         }
255         Objects.requireNonNull(temporal, "temporal");
256         try {
257             if (IsoChronology.INSTANCE.equals(Chronology.from(temporal)) == false) {
258                 temporal = LocalDate.from(temporal);
259             }
260             return of(temporal.get(YEAR), temporal.get(MONTH_OF_YEAR));
261         } catch (DateTimeException ex) {
262             throw new DateTimeException("Unable to obtain YearMonth from TemporalAccessor: " +
263                 temporal + " of type " + temporal.getClass().getName(), ex);
264         }
265     }
266
267     //-----
268     /**
269     * Obtains an instance of {@code YearMonth} from a text string such as {@code 2007-12}.
270     * <p>
271     * The string must represent a valid year-month.
272     * The format must be {@code uuuu-MM}.
273     * Years outside the range 0000 to 9999 must be prefixed by the plus or minus symbol.
274     *
275     * @param text the text to parse such as "2007-12", not null
276     * @return the parsed year-month, not null
277     * @throws DateTimeParseException if the text cannot be parsed
278     */
279     public static YearMonth parse(CharSequence text) {
280         return parse(text, PARSER);
281     }
282
283     /**
284     * Obtains an instance of {@code YearMonth} from a text string using a specific formatter.
285     * <p>
286     * The text is parsed using the formatter, returning a year-month.
287     *
288     * @param text the text to parse, not null
289     * @param formatter the formatter to use, not null
290     * @return the parsed year-month, not null
291     * @throws DateTimeParseException if the text cannot be parsed
292     */
293     public static YearMonth parse(CharSequence text, DateTimeFormatter formatter) {
294         Objects.requireNonNull(formatter, "formatter");
295         return formatter.parse(text, YearMonth::from);
296     }
297
298     //-----
299     /**
300     * Constructor.
301     *

```



```

302     * @param year the year to represent, validated from MIN_YEAR to MAX_YEAR
303     * @param month the month-of-year to represent, validated from 1 (January) to 12 (December)
304     */
305     private YearMonth(int year, int month) {
306         this.year = year;
307         this.month = month;
308     }
309
310     /**
311     * Returns a copy of this year-month with the new year and month, checking
312     * to see if a new object is in fact required.
313     *
314     * @param newYear the year to represent, validated from MIN_YEAR to MAX_YEAR
315     * @param newMonth the month-of-year to represent, validated not null
316     * @return the year-month, not null
317     */
318     private YearMonth with(int newYear, int newMonth) {
319         if (year == newYear && month == newMonth) {
320             return this;
321         }
322         return new YearMonth(newYear, newMonth);
323     }
324
325     //-----
326     /**
327     * Checks if the specified field is supported.
328     * <p>
329     * This checks if this year-month can be queried for the specified field.
330     * If false, then calling the {@link #range(TemporalField) range},
331     * {@link #get(TemporalField) get} and {@link #with(TemporalField, long)}
332     * methods will throw an exception.
333     * <p>
334     * If the field is a {@link ChronoField} then the query is implemented here.
335     * The supported fields are:
336     * <ul>
337     * <li>{@code MONTH_OF_YEAR}
338     * <li>{@code PROLEPTIC_MONTH}
339     * <li>{@code YEAR_OF_ERA}
340     * <li>{@code YEAR}
341     * <li>{@code ERA}
342     * </ul>
343     * All other {@code ChronoField} instances will return false.
344     * <p>
345     * If the field is not a {@code ChronoField}, then the result of this method
346     * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
347     * passing {@code this} as the argument.
348     * Whether the field is supported is determined by the field.
349     *
350     * @param field the field to check, null returns false
351     * @return true if the field is supported on this year-month, false if not
352     */
353     @Override
354     public boolean isSupported(TemporalField field) {

```

```

355     if (field instanceof ChronoField) {
356         return field == YEAR || field == MONTH_OF_YEAR ||
357             field == PROLEPTIC_MONTH || field == YEAR_OF_ERA || field == ERA;
358     }
359     return field != null && field.isSupportedBy(this);
360 }
361
362 /**
363  * Checks if the specified unit is supported.
364  * <p>
365  * This checks if the specified unit can be added to, or subtracted from, this year-month.
366  * If false, then calling the {@link #plus(long, TemporalUnit)} and
367  * {@link #minus(long, TemporalUnit) minus} methods will throw an exception.
368  * <p>
369  * If the unit is a {@link ChronoUnit} then the query is implemented here.
370  * The supported units are:
371  * <ul>
372  * <li>{@code MONTHS}
373  * <li>{@code YEARS}
374  * <li>{@code DECADES}
375  * <li>{@code CENTURIES}
376  * <li>{@code MILLENNIA}
377  * <li>{@code ERAS}
378  * </ul>
379  * All other {@code ChronoUnit} instances will return false.
380  * <p>
381  * If the unit is not a {@code ChronoUnit}, then the result of this method
382  * is obtained by invoking {@code TemporalUnit.isSupportedBy(Temporal)}
383  * passing {@code this} as the argument.
384  * Whether the unit is supported is determined by the unit.
385  *
386  * @param unit the unit to check, null returns false
387  * @return true if the unit can be added/subtracted, false if not
388  */
389 @Override
390 public boolean isSupported(TemporalUnit unit) {
391     if (unit instanceof ChronoUnit) {
392         return unit == MONTHS || unit == YEARS || unit == DECADES || unit == CENTURIES || unit
393             == MILLENNIA || unit == ERAS;
394     }
395     return unit != null && unit.isSupportedBy(this);
396 }
397
398 //-----
399 /**
400  * Gets the range of valid values for the specified field.
401  * <p>
402  * The range object expresses the minimum and maximum valid values for a field.
403  * This year-month is used to enhance the accuracy of the returned range.
404  * If it is not possible to return the range, because the field is not supported
405  * or for some other reason, an exception is thrown.
406  * <p>
407  * If the field is a {@link ChronoField} then the query is implemented here.

```

```

407     * The {@link #isSupported(TemporalField) supported fields} will return
408     * appropriate range instances.
409     * All other {@code ChronoField} instances will throw an {@code
      UnsupportedTemporalTypeException}.
410     * <p>
411     * If the field is not a {@code ChronoField}, then the result of this method
412     * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
413     * passing {@code this} as the argument.
414     * Whether the range can be obtained is determined by the field.
415     *
416     * @param field the field to query the range for, not null
417     * @return the range of valid values for the field, not null
418     * @throws DateTimeException if the range for the field cannot be obtained
419     * @throws UnsupportedTemporalTypeException if the field is not supported
420     */
421     @Override
422     public ValueRange range(TemporalField field) {
423         if (field == YEAR_OF_ERA) {
424             return (getYear() <= 0 ? ValueRange.of(1, Year.MAX_VALUE + 1) : ValueRange.of(1, Year.
              MAX_VALUE));
425         }
426         return Temporal.super.range(field);
427     }
428
429     /**
430     * Gets the value of the specified field from this year-month as an {@code int}.
431     * <p>
432     * This queries this year-month for the value of the specified field.
433     * The returned value will always be within the valid range of values for the field.
434     * If it is not possible to return the value, because the field is not supported
435     * or for some other reason, an exception is thrown.
436     * <p>
437     * If the field is a {@link ChronoField} then the query is implemented here.
438     * The {@link #isSupported(TemporalField) supported fields} will return valid
439     * values based on this year-month, except {@code PROLEPTIC_MONTH} which is too
440     * large to fit in an {@code int} and throw a {@code DateTimeException}.
441     * All other {@code ChronoField} instances will throw an {@code
      UnsupportedTemporalTypeException}.
442     * <p>
443     * If the field is not a {@code ChronoField}, then the result of this method
444     * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
445     * passing {@code this} as the argument. Whether the value can be obtained,
446     * and what the value represents, is determined by the field.
447     *
448     * @param field the field to get, not null
449     * @return the value for the field
450     * @throws DateTimeException if a value for the field cannot be obtained or
451     *         the value is outside the range of valid values for the field
452     * @throws UnsupportedTemporalTypeException if the field is not supported or
453     *         the range of values exceeds an {@code int}
454     * @throws ArithmeticException if numeric overflow occurs
455     */
456     @Override // override for Javadoc

```

```

457 public int get(TemporalField field) {
458     return range(field).checkValidIntValue(getLong(field), field);
459 }
460
461 /**
462  * Gets the value of the specified field from this year-month as a {@code long}.
463  * <p>
464  * This queries this year-month for the value of the specified field.
465  * If it is not possible to return the value, because the field is not supported
466  * or for some other reason, an exception is thrown.
467  * <p>
468  * If the field is a {@link ChronoField} then the query is implemented here.
469  * The {@link #isSupported(TemporalField) supported fields} will return valid
470  * values based on this year-month.
471  * All other {@code ChronoField} instances will throw an {@code
472     UnsupportedTemporalTypeException}.
473  * <p>
474  * If the field is not a {@code ChronoField}, then the result of this method
475  * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
476  * passing {@code this} as the argument. Whether the value can be obtained,
477  * and what the value represents, is determined by the field.
478  *
479  * @param field the field to get, not null
480  * @return the value for the field
481  * @throws DateTimeException if a value for the field cannot be obtained
482  * @throws UnsupportedTemporalTypeException if the field is not supported
483  * @throws ArithmeticException if numeric overflow occurs
484  */
485 @Override
486 public long getLong(TemporalField field) {
487     if (field instanceof ChronoField) {
488         switch ((ChronoField) field) {
489             case MONTH_OF_YEAR: return month;
490             case PROLEPTIC_MONTH: return getProlepticMonth();
491             case YEAR_OF_ERA: return (year < 1 ? 1 - year : year);
492             case YEAR: return year;
493             case ERA: return (year < 1 ? 0 : 1);
494         }
495         throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
496     }
497     return field.getFrom(this);
498 }
499
500 private long getProlepticMonth() {
501     return (year * 12L + month - 1);
502 }
503
504 //-----
505 /**
506  * Gets the year field.
507  * <p>
508  * This method returns the primitive {@code int} value for the year.
509  * <p>

```

```

509     * The year returned by this method is proleptic as per {@code get(YEAR)}.
510     *
511     * @return the year, from MIN_YEAR to MAX_YEAR
512     */
513     public int getYear() {
514         return year;
515     }
516
517     /**
518     * Gets the month-of-year field from 1 to 12.
519     * <p>
520     * This method returns the month as an {@code int} from 1 to 12.
521     * Application code is frequently clearer if the enum {@link Month}
522     * is used by calling {@link #getMonth()}.
523     *
524     * @return the month-of-year, from 1 to 12
525     * @see #getMonth()
526     */
527     public int getMonthValue() {
528         return month;
529     }
530
531     /**
532     * Gets the month-of-year field using the {@code Month} enum.
533     * <p>
534     * This method returns the enum {@link Month} for the month.
535     * This avoids confusion as to what {@code int} values mean.
536     * If you need access to the primitive {@code int} value then the enum
537     * provides the {@link Month#getValue() int value}.
538     *
539     * @return the month-of-year, not null
540     * @see #getMonthValue()
541     */
542     public Month getMonth() {
543         return Month.of(month);
544     }
545
546     //-----------------------------------------------------
547     /**
548     * Checks if the year is a leap year, according to the ISO proleptic
549     * calendar system rules.
550     * <p>
551     * This method applies the current rules for leap years across the whole time-line.
552     * In general, a year is a leap year if it is divisible by four without
553     * remainder. However, years divisible by 100, are not leap years, with
554     * the exception of years divisible by 400 which are.
555     * <p>
556     * For example, 1904 is a leap year it is divisible by 4.
557     * 1900 was not a leap year as it is divisible by 100, however 2000 was a
558     * leap year as it is divisible by 400.
559     * <p>
560     * The calculation is proleptic - applying the same rules into the far future and far past.
561     * This is historically inaccurate, but is correct for the ISO-8601 standard.

```

```

562     *
563     * @return true if the year is leap, false otherwise
564     */
565     public boolean isLeapYear() {
566         return IsoChronology.INSTANCE.isLeapYear(year);
567     }
568
569     /**
570     * Checks if the day-of-month is valid for this year-month.
571     * <p>
572     * This method checks whether this year and month and the input day form
573     * a valid date.
574     *
575     * @param dayOfMonth the day-of-month to validate, from 1 to 31, invalid value returns false
576     * @return true if the day is valid for this year-month
577     */
578     public boolean isValidDay(int dayOfMonth) {
579         return dayOfMonth >= 1 && dayOfMonth <= lengthOfMonth();
580     }
581
582     /**
583     * Returns the length of the month, taking account of the year.
584     * <p>
585     * This returns the length of the month in days.
586     * For example, a date in January would return 31.
587     *
588     * @return the length of the month in days, from 28 to 31
589     */
590     public int lengthOfMonth() {
591         return getMonth().length(isLeapYear());
592     }
593
594     /**
595     * Returns the length of the year.
596     * <p>
597     * This returns the length of the year in days, either 365 or 366.
598     *
599     * @return 366 if the year is leap, 365 otherwise
600     */
601     public int lengthOfYear() {
602         return (isLeapYear() ? 366 : 365);
603     }
604
605     //-----
606     /**
607     * Returns an adjusted copy of this year-month.
608     * <p>
609     * This returns a {@code YearMonth}, based on this one, with the year-month adjusted.
610     * The adjustment takes place using the specified adjuster strategy object.
611     * Read the documentation of the adjuster to understand what adjustment will be made.
612     * <p>
613     * A simple adjuster might simply set the one of the fields, such as the year field.
614     * A more complex adjuster might set the year-month to the next month that

```

```

615     * Halley's comet will pass the Earth.
616     * <p>
617     * The result of this method is obtained by invoking the
618     * {@link TemporalAdjuster#adjustInto(Temporal)} method on the
619     * specified adjuster passing {@code this} as the argument.
620     * <p>
621     * This instance is immutable and unaffected by this method call.
622     *
623     * @param adjuster the adjuster to use, not null
624     * @return a {@code YearMonth} based on {@code this} with the adjustment made, not null
625     * @throws DateTimeException if the adjustment cannot be made
626     * @throws ArithmeticException if numeric overflow occurs
627     */
628     @Override
629     public YearMonth with(TemporalAdjuster adjuster) {
630         return (YearMonth) adjuster.adjustInto(this);
631     }
632
633     /**
634     * Returns a copy of this year-month with the specified field set to a new value.
635     * <p>
636     * This returns a {@code YearMonth}, based on this one, with the value
637     * for the specified field changed.
638     * This can be used to change any supported field, such as the year or month.
639     * If it is not possible to set the value, because the field is not supported or for
640     * some other reason, an exception is thrown.
641     * <p>
642     * If the field is a {@link ChronoField} then the adjustment is implemented here.
643     * The supported fields behave as follows:
644     * <ul>
645     * <li>{@code MONTH_OF_YEAR} -
646     * Returns a {@code YearMonth} with the specified month-of-year.
647     * The year will be unchanged.
648     * <li>{@code PROLEPTIC_MONTH} -
649     * Returns a {@code YearMonth} with the specified proleptic-month.
650     * This completely replaces the year and month of this object.
651     * <li>{@code YEAR_OF_ERA} -
652     * Returns a {@code YearMonth} with the specified year-of-era
653     * The month and era will be unchanged.
654     * <li>{@code YEAR} -
655     * Returns a {@code YearMonth} with the specified year.
656     * The month will be unchanged.
657     * <li>{@code ERA} -
658     * Returns a {@code YearMonth} with the specified era.
659     * The month and year-of-era will be unchanged.
660     * </ul>
661     * <p>
662     * In all cases, if the new value is outside the valid range of values for the field
663     * then a {@code DateTimeException} will be thrown.
664     * <p>
665     * All other {@code ChronoField} instances will throw an {@code
        UnsupportedTemporalTypeException}.
666     * <p>

```

```

667     * If the field is not a {@code ChronoField}, then the result of this method
668     * is obtained by invoking {@code TemporalField.adjustInto(Temporal, long)}
669     * passing {@code this} as the argument. In this case, the field determines
670     * whether and how to adjust the instant.
671     * <p>
672     * This instance is immutable and unaffected by this method call.
673     *
674     * @param field the field to set in the result, not null
675     * @param newValue the new value of the field in the result
676     * @return a {@code YearMonth} based on {@code this} with the specified field set, not null
677     * @throws DateTimeException if the field cannot be set
678     * @throws UnsupportedTemporalTypeException if the field is not supported
679     * @throws ArithmeticException if numeric overflow occurs
680     */
681     @Override
682     public YearMonth with(TemporalField field, long newValue) {
683         if (field instanceof ChronoField) {
684             ChronoField f = (ChronoField) field;
685             f.checkValidValue(newValue);
686             switch (f) {
687                 case MONTH_OF_YEAR: return withMonth((int) newValue);
688                 case PROLEPTIC_MONTH: return plusMonths(newValue - getProlepticMonth());
689                 case YEAR_OF_ERA: return withYear((int) (year < 1 ? 1 - newValue : newValue));
690                 case YEAR: return withYear((int) newValue);
691                 case ERA: return (getLong(ERA) == newValue ? this : withYear(1 - year));
692             }
693             throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
694         }
695         return field.adjustInto(this, newValue);
696     }
697
698     //-----
699     /**
700     * Returns a copy of this {@code YearMonth} with the year altered.
701     * <p>
702     * This instance is immutable and unaffected by this method call.
703     *
704     * @param year the year to set in the returned year-month, from MIN_YEAR to MAX_YEAR
705     * @return a {@code YearMonth} based on this year-month with the requested year, not null
706     * @throws DateTimeException if the year value is invalid
707     */
708     public YearMonth withYear(int year) {
709         YEAR.checkValidValue(year);
710         return with(year, month);
711     }
712
713     /**
714     * Returns a copy of this {@code YearMonth} with the month-of-year altered.
715     * <p>
716     * This instance is immutable and unaffected by this method call.
717     *
718     * @param month the month-of-year to set in the returned year-month, from 1 (January) to 12 (
       December)

```



```

719     * @return a {@code YearMonth} based on this year-month with the requested month, not null
720     * @throws DateTimeException if the month-of-year value is invalid
721     */
722     public YearMonth withMonth(int month) {
723         MONTH_OF_YEAR.checkValidValue(month);
724         return with(year, month);
725     }
726
727     //-----
728     /**
729     * Returns a copy of this year-month with the specified amount added.
730     * <p>
731     * This returns a {@code YearMonth}, based on this one, with the specified amount added.
732     * The amount is typically {@link Period} but may be any other type implementing
733     * the {@link TemporalAmount} interface.
734     * <p>
735     * The calculation is delegated to the amount object by calling
736     * {@link TemporalAmount#addTo(Temporal)}. The amount implementation is free
737     * to implement the addition in any way it wishes, however it typically
738     * calls back to {@link #plus(long, TemporalUnit)}. Consult the documentation
739     * of the amount implementation to determine if it can be successfully added.
740     * <p>
741     * This instance is immutable and unaffected by this method call.
742     *
743     * @param amountToAdd the amount to add, not null
744     * @return a {@code YearMonth} based on this year-month with the addition made, not null
745     * @throws DateTimeException if the addition cannot be made
746     * @throws ArithmeticException if numeric overflow occurs
747     */
748     @Override
749     public YearMonth plus(TemporalAmount amountToAdd) {
750         return (YearMonth) amountToAdd.addTo(this);
751     }
752
753     /**
754     * Returns a copy of this year-month with the specified amount added.
755     * <p>
756     * This returns a {@code YearMonth}, based on this one, with the amount
757     * in terms of the unit added. If it is not possible to add the amount, because the
758     * unit is not supported or for some other reason, an exception is thrown.
759     * <p>
760     * If the field is a {@link ChronoUnit} then the addition is implemented here.
761     * The supported fields behave as follows:
762     * <ul>
763     * <li>{@code MONTHS} -
764     * Returns a {@code YearMonth} with the specified number of months added.
765     * This is equivalent to {@link #plusMonths(long)}.
766     * <li>{@code YEARS} -
767     * Returns a {@code YearMonth} with the specified number of years added.
768     * This is equivalent to {@link #plusYears(long)}.
769     * <li>{@code DECADES} -
770     * Returns a {@code YearMonth} with the specified number of decades added.
771     * This is equivalent to calling {@link #plusYears(long)} with the amount

```

```

772 * multiplied by 10.
773 * <li>{@code CENTURIES} -
774 * Returns a {@code YearMonth} with the specified number of centuries added.
775 * This is equivalent to calling {@link #plusYears(long)} with the amount
776 * multiplied by 100.
777 * <li>{@code MILLENNIA} -
778 * Returns a {@code YearMonth} with the specified number of millennia added.
779 * This is equivalent to calling {@link #plusYears(long)} with the amount
780 * multiplied by 1,000.
781 * <li>{@code ERAS} -
782 * Returns a {@code YearMonth} with the specified number of eras added.
783 * Only two eras are supported so the amount must be one, zero or minus one.
784 * If the amount is non-zero then the year is changed such that the year-of-era
785 * is unchanged.
786 * </ul>
787 * <p>
788 * All other {@code ChronoUnit} instances will throw an {@code UnsupportedOperationException}
789 *   }.
790 * <p>
791 * If the field is not a {@code ChronoUnit}, then the result of this method
792 * is obtained by invoking {@code TemporalUnit.addTo(Temporal, long)}
793 * passing {@code this} as the argument. In this case, the unit determines
794 * whether and how to perform the addition.
795 * <p>
796 * This instance is immutable and unaffected by this method call.
797 *
798 * @param amountToAdd the amount of the unit to add to the result, may be negative
799 * @param unit the unit of the amount to add, not null
800 * @return a {@code YearMonth} based on this year-month with the specified amount added, not
801 *   null
802 * @throws DateTimeException if the addition cannot be made
803 * @throws UnsupportedOperationException if the unit is not supported
804 * @throws ArithmeticException if numeric overflow occurs
805 */
806 @Override
807 public YearMonth plus(long amountToAdd, TemporalUnit unit) {
808     if (unit instanceof ChronoUnit) {
809         switch ((ChronoUnit) unit) {
810             case MONTHS: return plusMonths(amountToAdd);
811             case YEARS: return plusYears(amountToAdd);
812             case DECADES: return plusYears(Math.multiplyExact(amountToAdd, 10));
813             case CENTURIES: return plusYears(Math.multiplyExact(amountToAdd, 100));
814             case MILLENNIA: return plusYears(Math.multiplyExact(amountToAdd, 1000));
815             case ERAS: return with(ERA, Math.addExact(getLong(ERA), amountToAdd));
816         }
817         throw new UnsupportedOperationException("Unsupported unit: " + unit);
818     }
819     return unit.addTo(this, amountToAdd);
820 }
821 /**
822 * Returns a copy of this {@code YearMonth} with the specified number of years added.
823 * <p>

```

```

823     * This instance is immutable and unaffected by this method call.
824     *
825     * @param yearsToAdd the years to add, may be negative
826     * @return a {@code YearMonth} based on this year-month with the years added, not null
827     * @throws DateTimeException if the result exceeds the supported range
828     */
829     public YearMonth plusYears(long yearsToAdd) {
830         if (yearsToAdd == 0) {
831             return this;
832         }
833         int newYear = YEAR.checkValidIntValue(year + yearsToAdd); // safe overflow
834         return with(newYear, month);
835     }
836
837     /**
838     * Returns a copy of this {@code YearMonth} with the specified number of months added.
839     * <p>
840     * This instance is immutable and unaffected by this method call.
841     *
842     * @param monthsToAdd the months to add, may be negative
843     * @return a {@code YearMonth} based on this year-month with the months added, not null
844     * @throws DateTimeException if the result exceeds the supported range
845     */
846     public YearMonth plusMonths(long monthsToAdd) {
847         if (monthsToAdd == 0) {
848             return this;
849         }
850         long monthCount = year * 12L + (month - 1);
851         long calcMonths = monthCount + monthsToAdd; // safe overflow
852         int newYear = YEAR.checkValidIntValue(Math.floorDiv(calcMonths, 12));
853         int newMonth = (int) Math.floorMod(calcMonths, 12) + 1;
854         return with(newYear, newMonth);
855     }
856
857     //-----
858     /**
859     * Returns a copy of this year-month with the specified amount subtracted.
860     * <p>
861     * This returns a {@code YearMonth}, based on this one, with the specified amount subtracted.
862     * The amount is typically {@link Period} but may be any other type implementing
863     * the {@link TemporalAmount} interface.
864     * <p>
865     * The calculation is delegated to the amount object by calling
866     * {@link TemporalAmount#subtractFrom(Temporal)}. The amount implementation is free
867     * to implement the subtraction in any way it wishes, however it typically
868     * calls back to {@link #minus(long, TemporalUnit)}. Consult the documentation
869     * of the amount implementation to determine if it can be successfully subtracted.
870     * <p>
871     * This instance is immutable and unaffected by this method call.
872     *
873     * @param amountToSubtract the amount to subtract, not null
874     * @return a {@code YearMonth} based on this year-month with the subtraction made, not null
875     * @throws DateTimeException if the subtraction cannot be made

```

```

876     * @throws ArithmeticException if numeric overflow occurs
877     */
878     @Override
879     public YearMonth minus(TemporalAmount amountToSubtract) {
880         return (YearMonth) amountToSubtract.subtractFrom(this);
881     }
882
883     /**
884     * Returns a copy of this year-month with the specified amount subtracted.
885     * <p>
886     * This returns a {@code YearMonth}, based on this one, with the amount
887     * in terms of the unit subtracted. If it is not possible to subtract the amount,
888     * because the unit is not supported or for some other reason, an exception is thrown.
889     * <p>
890     * This method is equivalent to {@link #plus(long, TemporalUnit)} with the amount negated.
891     * See that method for a full description of how addition, and thus subtraction, works.
892     * <p>
893     * This instance is immutable and unaffected by this method call.
894     *
895     * @param amountToSubtract the amount of the unit to subtract from the result, may be negative
896     * @param unit the unit of the amount to subtract, not null
897     * @return a {@code YearMonth} based on this year-month with the specified amount subtracted,
898     *         not null
899     * @throws DateTimeException if the subtraction cannot be made
900     * @throws UnsupportedTemporalTypeException if the unit is not supported
901     * @throws ArithmeticException if numeric overflow occurs
902     */
903     @Override
904     public YearMonth minus(long amountToSubtract, TemporalUnit unit) {
905         return (amountToSubtract == Long.MIN_VALUE ? plus(Long.MAX_VALUE, unit).plus(1, unit) :
906             plus(-amountToSubtract, unit));
907     }
908
909     /**
910     * Returns a copy of this {@code YearMonth} with the specified number of years subtracted.
911     * <p>
912     * This instance is immutable and unaffected by this method call.
913     *
914     * @param yearsToSubtract the years to subtract, may be negative
915     * @return a {@code YearMonth} based on this year-month with the years subtracted, not null
916     * @throws DateTimeException if the result exceeds the supported range
917     */
918     public YearMonth minusYears(long yearsToSubtract) {
919         return (yearsToSubtract == Long.MIN_VALUE ? plusYears(Long.MAX_VALUE).plusYears(1) :
920             plusYears(-yearsToSubtract));
921     }
922
923     /**
924     * Returns a copy of this {@code YearMonth} with the specified number of months subtracted.
925     * <p>
926     * This instance is immutable and unaffected by this method call.
927     *
928     * @param monthsToSubtract the months to subtract, may be negative

```

```

926     * @return a {@code YearMonth} based on this year-month with the months subtracted, not null
927     * @throws DateTimeException if the result exceeds the supported range
928     */
929     public YearMonth minusMonths(long monthsToSubtract) {
930         return (monthsToSubtract == Long.MIN_VALUE ? plusMonths(Long.MAX_VALUE).plusMonths(1) :
931             plusMonths(-monthsToSubtract));
932     }
933
934     /**
935      * Queries this year-month using the specified query.
936      * <p>
937      * This queries this year-month using the specified query strategy object.
938      * The {@code TemporalQuery} object defines the logic to be used to
939      * obtain the result. Read the documentation of the query to understand
940      * what the result of this method will be.
941      * <p>
942      * The result of this method is obtained by invoking the
943      * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
944      * specified query passing {@code this} as the argument.
945      *
946      * @param <R> the type of the result
947      * @param query the query to invoke, not null
948      * @return the query result, null may be returned (defined by the query)
949      * @throws DateTimeException if unable to query (defined by the query)
950      * @throws ArithmeticException if numeric overflow occurs (defined by the query)
951      */
952     @SuppressWarnings("unchecked")
953     @Override
954     public <R> query(TemporalQuery<R> query) {
955         if (query == TemporalQueries.chronology()) {
956             return (R) IsoChronology.INSTANCE;
957         } else if (query == TemporalQueries.precision()) {
958             return (R) MONTHS;
959         }
960         return Temporal.super.query(query);
961     }
962
963     /**
964      * Adjusts the specified temporal object to have this year-month.
965      * <p>
966      * This returns a temporal object of the same observable type as the input
967      * with the year and month changed to be the same as this.
968      * <p>
969      * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
970      * passing {@link ChronoField#PROLEPTIC_MONTH} as the field.
971      * If the specified temporal object does not use the ISO calendar system then
972      * a {@code DateTimeException} is thrown.
973      * <p>
974      * In most cases, it is clearer to reverse the calling pattern by using
975      * {@link Temporal#with(TemporalAdjuster)}:
976      * <pre>
977      * // these two lines are equivalent, but the second approach is recommended

```

```

978     * temporal = thisYearMonth.adjustInto(temporal);
979     * temporal = temporal.with(thisYearMonth);
980     * </pre>
981     * <p>
982     * This instance is immutable and unaffected by this method call.
983     *
984     * @param temporal the target object to be adjusted, not null
985     * @return the adjusted object, not null
986     * @throws DateTimeException if unable to make the adjustment
987     * @throws ArithmeticException if numeric overflow occurs
988     */
989     @Override
990     public Temporal adjustInto(Temporal temporal) {
991         if (Chronology.from(temporal).equals(IsoChronology.INSTANCE) == false) {
992             throw new DateTimeException("Adjustment only supported on ISO date-time");
993         }
994         return temporal.with(PROLEPTIC_MONTH, getProlepticMonth());
995     }
996
997     /**
998     * Calculates the amount of time until another year-month in terms of the specified unit.
999     * <p>
1000     * This calculates the amount of time between two {@code YearMonth}
1001     * objects in terms of a single {@code TemporalUnit}.
1002     * The start and end points are {@code this} and the specified year-month.
1003     * The result will be negative if the end is before the start.
1004     * The {@code Temporal} passed to this method is converted to a
1005     * {@code YearMonth} using {@link #from(TemporalAccessor)}.
1006     * For example, the amount in years between two year-months can be calculated
1007     * using {@code startYearMonth.until(endYearMonth, YEARS)}.
1008     * <p>
1009     * The calculation returns a whole number, representing the number of
1010     * complete units between the two year-months.
1011     * For example, the amount in decades between 2012-06 and 2032-05
1012     * will only be one decade as it is one month short of two decades.
1013     * <p>
1014     * There are two equivalent ways of using this method.
1015     * The first is to invoke this method.
1016     * The second is to use {@link TemporalUnit#between(Temporal, Temporal)}:
1017     * <pre>
1018     * // these two lines are equivalent
1019     * amount = start.until(end, MONTHS);
1020     * amount = MONTHS.between(start, end);
1021     * </pre>
1022     * The choice should be made based on which makes the code more readable.
1023     * <p>
1024     * The calculation is implemented in this method for {@link ChronoUnit}.
1025     * The units {@code MONTHS}, {@code YEARS}, {@code DECADES},
1026     * {@code CENTURIES}, {@code MILLENNIA} and {@code ERAS} are supported.
1027     * Other {@code ChronoUnit} values will throw an exception.
1028     * <p>
1029     * If the unit is not a {@code ChronoUnit}, then the result of this method
1030     * is obtained by invoking {@code TemporalUnit.between(Temporal, Temporal)}

```

```

1031     * passing {@code this} as the first argument and the converted input temporal
1032     * as the second argument.
1033     * <p>
1034     * This instance is immutable and unaffected by this method call.
1035     *
1036     * @param endExclusive the end date, exclusive, which is converted to a {@code YearMonth}, not
1037         null
1038     * @param unit the unit to measure the amount in, not null
1039     * @return the amount of time between this year-month and the end year-month
1040     * @throws DateTimeException if the amount cannot be calculated, or the end
1041     *     temporal cannot be converted to a {@code YearMonth}
1042     * @throws UnsupportedTemporalTypeException if the unit is not supported
1043     * @throws ArithmeticException if numeric overflow occurs
1044     */
1045     @Override
1046     public long until(Temporal endExclusive, TemporalUnit unit) {
1047         YearMonth end = YearMonth.from(endExclusive);
1048         if (unit instanceof ChronoUnit) {
1049             long monthsUntil = end.getProlepticMonth() - getProlepticMonth(); // no overflow
1050             switch ((ChronoUnit) unit) {
1051                 case MONTHS: return monthsUntil;
1052                 case YEARS: return monthsUntil / 12;
1053                 case DECADES: return monthsUntil / 120;
1054                 case CENTURIES: return monthsUntil / 1200;
1055                 case MILLENNIA: return monthsUntil / 12000;
1056                 case ERAS: return end.getLong(ERA) - getLong(ERA);
1057             }
1058             throw new UnsupportedTemporalTypeException("Unsupported unit: " + unit);
1059         }
1060         return unit.between(this, end);
1061     }
1062
1063     /**
1064     * Formats this year-month using the specified formatter.
1065     * <p>
1066     * This year-month will be passed to the formatter to produce a string.
1067     *
1068     * @param formatter the formatter to use, not null
1069     * @return the formatted year-month string, not null
1070     * @throws DateTimeException if an error occurs during printing
1071     */
1072     public String format(DateTimeFormatter formatter) {
1073         Objects.requireNonNull(formatter, "formatter");
1074         return formatter.format(this);
1075     }
1076
1077     //-----
1078     /**
1079     * Combines this year-month with a day-of-month to create a {@code LocalDate}.
1080     * <p>
1081     * This returns a {@code LocalDate} formed from this year-month and the specified day-of-month.
1082     * <p>
1083     * The day-of-month value must be valid for the year-month.

```



```

1083     * <p>
1084     * This method can be used as part of a chain to produce a date:
1085     * <pre>
1086     *   LocalDate date = year.atMonth(month).atDay(day);
1087     * </pre>
1088     *
1089     * @param dayOfMonth the day-of-month to use, from 1 to 31
1090     * @return the date formed from this year-month and the specified day, not null
1091     * @throws DateTimeException if the day is invalid for the year-month
1092     * @see #isValidDay(int)
1093     */
1094     public LocalDate atDay(int dayOfMonth) {
1095         return LocalDate.of(year, month, dayOfMonth);
1096     }
1097
1098     /**
1099     * Returns a {@code LocalDate} at the end of the month.
1100     * <p>
1101     * This returns a {@code LocalDate} based on this year-month.
1102     * The day-of-month is set to the last valid day of the month, taking
1103     * into account leap years.
1104     * <p>
1105     * This method can be used as part of a chain to produce a date:
1106     * <pre>
1107     *   LocalDate date = year.atMonth(month).atEndOfMonth();
1108     * </pre>
1109     *
1110     * @return the last valid date of this year-month, not null
1111     */
1112     public LocalDate atEndOfMonth() {
1113         return LocalDate.of(year, month, lengthOfMonth());
1114     }
1115
1116     //-----
1117     /**
1118     * Compares this year-month to another year-month.
1119     * <p>
1120     * The comparison is based first on the value of the year, then on the value of the month.
1121     * It is "consistent with equals", as defined by {@link Comparable}.
1122     *
1123     * @param other the other year-month to compare to, not null
1124     * @return the comparator value, negative if less, positive if greater
1125     */
1126     @Override
1127     public int compareTo(YearMonth other) {
1128         int cmp = (year - other.year);
1129         if (cmp == 0) {
1130             cmp = (month - other.month);
1131         }
1132         return cmp;
1133     }
1134
1135     /**

```



```

1136     * Checks if this year-month is after the specified year-month.
1137     *
1138     * @param other the other year-month to compare to, not null
1139     * @return true if this is after the specified year-month
1140     */
1141     public boolean isAfter(YearMonth other) {
1142         return compareTo(other) > 0;
1143     }
1144
1145     /**
1146     * Checks if this year-month is before the specified year-month.
1147     *
1148     * @param other the other year-month to compare to, not null
1149     * @return true if this point is before the specified year-month
1150     */
1151     public boolean isBefore(YearMonth other) {
1152         return compareTo(other) < 0;
1153     }
1154
1155     //-----
1156     /**
1157     * Checks if this year-month is equal to another year-month.
1158     * <p>
1159     * The comparison is based on the time-line position of the year-months.
1160     *
1161     * @param obj the object to check, null returns false
1162     * @return true if this is equal to the other year-month
1163     */
1164     @Override
1165     public boolean equals(Object obj) {
1166         if (this == obj) {
1167             return true;
1168         }
1169         if (obj instanceof YearMonth) {
1170             YearMonth other = (YearMonth) obj;
1171             return year == other.year && month == other.month;
1172         }
1173         return false;
1174     }
1175
1176     /**
1177     * A hash code for this year-month.
1178     *
1179     * @return a suitable hash code
1180     */
1181     @Override
1182     public int hashCode() {
1183         return year ^ (month << 27);
1184     }
1185
1186     //-----
1187     /**
1188     * Outputs this year-month as a {@code String}, such as {@code 2007-12}.

```

```

1189     * <p>
1190     * The output will be in the format {@code uuuu-MM}:
1191     *
1192     * @return a string representation of this year-month, not null
1193     */
1194     @Override
1195     public String toString() {
1196         int absYear = Math.abs(year);
1197         StringBuilder buf = new StringBuilder(9);
1198         if (absYear < 1000) {
1199             if (year < 0) {
1200                 buf.append(year - 10000).deleteCharAt(1);
1201             } else {
1202                 buf.append(year + 10000).deleteCharAt(0);
1203             }
1204         } else {
1205             buf.append(year);
1206         }
1207         return buf.append(month < 10 ? "-0" : "-")
1208             .append(month)
1209             .toString();
1210     }
1211
1212     //-----
1213     /**
1214     * Writes the object using a
1215     * <a href="../..//serialized-form.html#java.time.Ser">dedicated serialized form</a>.
1216     * @serialData
1217     * <pre>
1218     *   out.writeByte(12); // identifies a YearMonth
1219     *   out.writeInt(year);
1220     *   out.writeByte(month);
1221     * </pre>
1222     *
1223     * @return the instance of {@code Ser}, not null
1224     */
1225     private Object writeReplace() {
1226         return new Ser(Ser.YEAR_MONTH_TYPE, this);
1227     }
1228
1229     /**
1230     * Defend against malicious streams.
1231     *
1232     * @param s the stream to read
1233     * @throws InvalidObjectException always
1234     */
1235     private void readObject(ObjectInputStream s) throws InvalidObjectException {
1236         throw new InvalidObjectException("Deserialization via serialization delegate");
1237     }
1238
1239     void writeExternal(DataOutput out) throws IOException {
1240         out.writeInt(year);
1241         out.writeByte(month);

```

```
1242     }
1243
1244     static YearMonth readExternal(DataInput in) throws IOException {
1245         int year = in.readInt();
1246         byte month = in.readByte();
1247         return YearMonth.of(year, month);
1248     }
1249
1250 }
```

4.17 ZoneId.java

Secuencia 66: java/time/ZoneId.java

```
1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
```

```
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time;
63
64  import java.io.DataOutput;
65  import java.io.IOException;
66  import java.io.InvalidObjectException;
67  import java.io.ObjectInputStream;
68  import java.io.Serializable;
69  import java.time.format.DateTimeFormatterBuilder;
70  import java.time.format.TextStyle;
71  import java.time.temporal.TemporalAccessor;
72  import java.time.temporal.TemporalField;
73  import java.time.temporal.TemporalQueries;
74  import java.time.temporal.TemporalQuery;
75  import java.time.temporal.UnsupportedTemporalTypeException;
76  import java.time.zone.ZoneRules;
77  import java.time.zone.ZoneRulesException;
78  import java.time.zone.ZoneRulesProvider;
79  import java.util.Collections;
80  import java.util.HashMap;
81  import java.util.Locale;
82  import java.util.Map;
83  import java.util.Objects;
84  import java.util.Set;
85  import java.util.TimeZone;
86
87  /**
88   * A time-zone ID, such as {@code Europe/Paris}.
89   * <p>
90   * A {@code ZoneId} is used to identify the rules used to convert between
91   * an {@link Instant} and a {@link LocalDateTime}.
92   * There are two distinct types of ID:
93   * <ul>
```

```
94 * <li>Fixed offsets - a fully resolved offset from UTC/Greenwich, that uses
95 * the same offset for all local date-times
96 * <li>Geographical regions - an area where a specific set of rules for finding
97 * the offset from UTC/Greenwich apply
98 * </ul>
99 * Most fixed offsets are represented by {@link ZoneOffset}.
100 * Calling {@link #normalized()} on any {@code ZoneId} will ensure that a
101 * fixed offset ID will be represented as a {@code ZoneOffset}.
102 * <p>
103 * The actual rules, describing when and how the offset changes, are defined by {@link ZoneRules}.
104 * This class is simply an ID used to obtain the underlying rules.
105 * This approach is taken because rules are defined by governments and change
106 * frequently, whereas the ID is stable.
107 * <p>
108 * The distinction has other effects. Serializing the {@code ZoneId} will only send
109 * the ID, whereas serializing the rules sends the entire data set.
110 * Similarly, a comparison of two IDs only examines the ID, whereas
111 * a comparison of two rules examines the entire data set.
112 *
113 * <h3>Time-zone IDs</h3>
114 * The ID is unique within the system.
115 * There are three types of ID.
116 * <p>
117 * The simplest type of ID is that from {@code ZoneOffset}.
118 * This consists of 'Z' and IDs starting with '+' or '-'.
119 * <p>
120 * The next type of ID are offset-style IDs with some form of prefix,
121 * such as 'GMT+2' or 'UTC+01:00'.
122 * The recognised prefixes are 'UTC', 'GMT' and 'UT'.
123 * The offset is the suffix and will be normalized during creation.
124 * These IDs can be normalized to a {@code ZoneOffset} using {@code normalized()}.
125 * <p>
126 * The third type of ID are region-based IDs. A region-based ID must be of
127 * two or more characters, and not start with 'UTC', 'GMT', 'UT' '+' or '-'.
128 * Region-based IDs are defined by configuration, see {@link ZoneRulesProvider}.
129 * The configuration focuses on providing the lookup from the ID to the
130 * underlying {@code ZoneRules}.
131 * <p>
132 * Time-zone rules are defined by governments and change frequently.
133 * There are a number of organizations, known here as groups, that monitor
134 * time-zone changes and collate them.
135 * The default group is the IANA Time Zone Database (TZDB).
136 * Other organizations include IATA (the airline industry body) and Microsoft.
137 * <p>
138 * Each group defines its own format for the region ID it provides.
139 * The TZDB group defines IDs such as 'Europe/London' or 'America/New_York'.
140 * TZDB IDs take precedence over other groups.
141 * <p>
142 * It is strongly recommended that the group name is included in all IDs supplied by
143 * groups other than TZDB to avoid conflicts. For example, IATA airline time-zone
144 * region IDs are typically the same as the three letter airport code.
145 * However, the airport of Utrecht has the code 'UTC', which is obviously a conflict.
146 * The recommended format for region IDs from groups other than TZDB is 'group-region'.
```

```

147 * Thus if IATA data were defined, Utrecht airport would be 'IATA~UTC'.
148 *
149 * <h3>Serialization</h3>
150 * This class can be serialized and stores the string zone ID in the external form.
151 * The {@code ZoneOffset} subclass uses a dedicated format that only stores the
152 * offset from UTC/Greenwich.
153 * <p>
154 * A {@code ZoneId} can be deserialized in a Java Runtime where the ID is unknown.
155 * For example, if a server-side Java Runtime has been updated with a new zone ID, but
156 * the client-side Java Runtime has not been updated. In this case, the {@code ZoneId}
157 * object will exist, and can be queried using {@code getId}, {@code equals},
158 * {@code hashCode}, {@code toString}, {@code getDisplayName} and {@code normalized}.
159 * However, any call to {@code getRules} will fail with {@code ZoneRulesException}.
160 * This approach is designed to allow a {@link ZonedDateTime} to be loaded and
161 * queried, but not modified, on a Java Runtime with incomplete time-zone information.
162 *
163 * <p>
164 * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
165 * class; use of identity-sensitive operations (including reference equality
166 * ({@code ==}), identity hash code, or synchronization) on instances of
167 * {@code ZoneId} may have unpredictable results and should be avoided.
168 * The {@code equals} method should be used for comparisons.
169 *
170 * @implSpec
171 * This abstract class has two implementations, both of which are immutable and thread-safe.
172 * One implementation models region-based IDs, the other is {@code ZoneOffset} modelling
173 * offset-based IDs. This difference is visible in serialization.
174 *
175 * @since 1.8
176 */
177 public abstract class ZoneId implements Serializable {
178
179     /**
180      * A map of zone overrides to enable the short time-zone names to be used.
181      * <p>
182      * Use of short zone IDs has been deprecated in {@code java.util.TimeZone}.
183      * This map allows the IDs to continue to be used via the
184      * {@link #of(String, Map)} factory method.
185      * <p>
186      * This map contains a mapping of the IDs that is in line with TZDB 2005r and
187      * later, where 'EST', 'MST' and 'HST' map to IDs which do not include daylight
188      * savings.
189      * <p>
190      * This maps as follows:
191      * <ul>
192      * <li>EST - -05:00</li>
193      * <li>HST - -10:00</li>
194      * <li>MST - -07:00</li>
195      * <li>ACT - Australia/Darwin</li>
196      * <li>AET - Australia/Sydney</li>
197      * <li>AGT - America/Argentina/Buenos_Aires</li>
198      * <li>ART - Africa/Cairo</li>
199      * <li>AST - America/Anchorage</li>

```

```

200 * <li>BET - America/Sao_Paulo</li>
201 * <li>BST - Asia/Dhaka</li>
202 * <li>CAT - Africa/Harare</li>
203 * <li>CNT - America/St_Johns</li>
204 * <li>CST - America/Chicago</li>
205 * <li>CTT - Asia/Shanghai</li>
206 * <li>EAT - Africa/Addis_Ababa</li>
207 * <li>ECT - Europe/Paris</li>
208 * <li>IET - America/Indiana/Indianapolis</li>
209 * <li>IST - Asia/Kolkata</li>
210 * <li>JST - Asia/Tokyo</li>
211 * <li>MIT - Pacific/Apia</li>
212 * <li>NET - Asia/Yerevan</li>
213 * <li>NST - Pacific/Auckland</li>
214 * <li>PLT - Asia/Karachi</li>
215 * <li>PNT - America/Phoenix</li>
216 * <li>PRT - America/Puerto_Rico</li>
217 * <li>PST - America/Los_Angeles</li>
218 * <li>SST - Pacific/Guadacanal</li>
219 * <li>VST - Asia/Ho_Chi_Minh</li>
220 * </ul>
221 * The map is unmodifiable.
222 */
223 public static final Map<String, String> SHORT_IDS;
224 static {
225     Map<String, String> map = new HashMap<>(64);
226     map.put("ACT", "Australia/Darwin");
227     map.put("AET", "Australia/Sydney");
228     map.put("AGT", "America/Argentina/Buenos_Aires");
229     map.put("ART", "Africa/Cairo");
230     map.put("AST", "America/Anchorage");
231     map.put("BET", "America/Sao_Paulo");
232     map.put("BST", "Asia/Dhaka");
233     map.put("CAT", "Africa/Harare");
234     map.put("CNT", "America/St_Johns");
235     map.put("CST", "America/Chicago");
236     map.put("CTT", "Asia/Shanghai");
237     map.put("EAT", "Africa/Addis_Ababa");
238     map.put("ECT", "Europe/Paris");
239     map.put("IET", "America/Indiana/Indianapolis");
240     map.put("IST", "Asia/Kolkata");
241     map.put("JST", "Asia/Tokyo");
242     map.put("MIT", "Pacific/Apia");
243     map.put("NET", "Asia/Yerevan");
244     map.put("NST", "Pacific/Auckland");
245     map.put("PLT", "Asia/Karachi");
246     map.put("PNT", "America/Phoenix");
247     map.put("PRT", "America/Puerto_Rico");
248     map.put("PST", "America/Los_Angeles");
249     map.put("SST", "Pacific/Guadacanal");
250     map.put("VST", "Asia/Ho_Chi_Minh");
251     map.put("EST", "-05:00");
252     map.put("MST", "-07:00");

```

```

253     map.put("HST", "-10:00");
254     SHORT_IDS = Collections.unmodifiableMap(map);
255 }
256 /**
257  * Serialization version.
258  */
259 private static final long serialVersionUID = 8352817235686L;
260
261 //-----
262 /**
263  * Gets the system default time-zone.
264  * <p>
265  * This queries {@link TimeZone#getDefault()} to find the default time-zone
266  * and converts it to a {@code ZoneId}. If the system default time-zone is changed,
267  * then the result of this method will also change.
268  *
269  * @return the zone ID, not null
270  * @throws DateTimeException if the converted zone ID has an invalid format
271  * @throws ZoneRulesException if the converted zone region ID cannot be found
272  */
273 public static ZoneId systemDefault() {
274     return TimeZone.getDefault().toZoneId();
275 }
276
277 /**
278  * Gets the set of available zone IDs.
279  * <p>
280  * This set includes the string form of all available region-based IDs.
281  * Offset-based zone IDs are not included in the returned set.
282  * The ID can be passed to {@link #of(String)} to create a {@code ZoneId}.
283  * <p>
284  * The set of zone IDs can increase over time, although in a typical application
285  * the set of IDs is fixed. Each call to this method is thread-safe.
286  *
287  * @return a modifiable copy of the set of zone IDs, not null
288  */
289 public static Set<String> getAvailableZoneIds() {
290     return ZoneRulesProvider.getAvailableZoneIds();
291 }
292
293 //-----
294 /**
295  * Obtains an instance of {@code ZoneId} using its ID using a map
296  * of aliases to supplement the standard zone IDs.
297  * <p>
298  * Many users of time-zones use short abbreviations, such as PST for
299  * 'Pacific Standard Time' and PDT for 'Pacific Daylight Time'.
300  * These abbreviations are not unique, and so cannot be used as IDs.
301  * This method allows a map of string to time-zone to be setup and reused
302  * within an application.
303  *
304  * @param zoneId the time-zone ID, not null
305  * @param aliasMap a map of alias zone IDs (typically abbreviations) to real zone IDs, not

```



```

    null
306 * @return the zone ID, not null
307 * @throws DateTimeException if the zone ID has an invalid format
308 * @throws ZoneRulesException if the zone ID is a region ID that cannot be found
309 */
310 public static ZoneId of(String zoneId, Map<String, String> aliasMap) {
311     Objects.requireNonNull(zoneId, "zoneId");
312     Objects.requireNonNull(aliasMap, "aliasMap");
313     String id = aliasMap.get(zoneId);
314     id = (id != null ? id : zoneId);
315     return of(id);
316 }
317
318 /**
319 * Obtains an instance of {@code ZoneId} from an ID ensuring that the
320 * ID is valid and available for use.
321 * <p>
322 * This method parses the ID producing a {@code ZoneId} or {@code ZoneOffset}.
323 * A {@code ZoneOffset} is returned if the ID is 'Z', or starts with '+' or '-'.
324 * The result will always be a valid ID for which {@link ZoneRules} can be obtained.
325 * <p>
326 * Parsing matches the zone ID step by step as follows.
327 * <ul>
328 * <li>If the zone ID equals 'Z', the result is {@code ZoneOffset.UTC}.
329 * <li>If the zone ID consists of a single letter, the zone ID is invalid
330 *   and {@code DateTimeException} is thrown.
331 * <li>If the zone ID starts with '+' or '-', the ID is parsed as a
332 *   {@code ZoneOffset} using {@link ZoneOffset#of(String)}.
333 * <li>If the zone ID equals 'GMT', 'UTC' or 'UT' then the result is a {@code ZoneId}
334 *   with the same ID and rules equivalent to {@code ZoneOffset.UTC}.
335 * <li>If the zone ID starts with 'UTC+', 'UTC-', 'GMT+', 'GMT-', 'UT+' or 'UT-'
336 *   then the ID is a prefixed offset-based ID. The ID is split in two, with
337 *   a two or three letter prefix and a suffix starting with the sign.
338 *   The suffix is parsed as a {@link ZoneOffset#of(String) ZoneOffset}.
339 *   The result will be a {@code ZoneId} with the specified UTC/GMT/UT prefix
340 *   and the normalized offset ID as per {@link ZoneOffset#getId()}.
341 *   The rules of the returned {@code ZoneId} will be equivalent to the
342 *   parsed {@code ZoneOffset}.
343 * <li>All other IDs are parsed as region-based zone IDs. Region IDs must
344 *   match the regular expression <code>[A-Za-z][A-Za-z0-9~/._-]+</code>
345 *   otherwise a {@code DateTimeException} is thrown. If the zone ID is not
346 *   in the configured set of IDs, {@code ZoneRulesException} is thrown.
347 *   The detailed format of the region ID depends on the group supplying the data.
348 *   The default set of data is supplied by the IANA Time Zone Database (TZDB).
349 *   This has region IDs of the form '{area}/{city}', such as 'Europe/Paris' or 'America/
350 *     New_York'.
351 *   This is compatible with most IDs from {@link java.util.TimeZone}.
352 * </ul>
353 *
354 * @param zoneId the time-zone ID, not null
355 * @return the zone ID, not null
356 * @throws DateTimeException if the zone ID has an invalid format
357 * @throws ZoneRulesException if the zone ID is a region ID that cannot be found

```

```

357     */
358     public static ZoneId of(String zoneId) {
359         return of(zoneId, true);
360     }
361
362     /**
363      * Obtains an instance of {@code ZoneId} wrapping an offset.
364      * <p>
365      * If the prefix is "GMT", "UTC", or "UT" a {@code ZoneId}
366      * with the prefix and the non-zero offset is returned.
367      * If the prefix is empty {@code ""} the {@code ZoneOffset} is returned.
368      *
369      * @param prefix the time-zone ID, not null
370      * @param offset the offset, not null
371      * @return the zone ID, not null
372      * @throws IllegalArgumentException if the prefix is not one of
373      *      "GMT", "UTC", or "UT", or ""
374      */
375     public static ZoneId ofOffset(String prefix, ZoneOffset offset) {
376         Objects.requireNonNull(prefix, "prefix");
377         Objects.requireNonNull(offset, "offset");
378         if (prefix.length() == 0) {
379             return offset;
380         }
381
382         if (!prefix.equals("GMT") && !prefix.equals("UTC") && !prefix.equals("UT")) {
383             throw new IllegalArgumentException("prefix should be GMT, UTC or UT, is: " + prefix);
384         }
385
386         if (offset.getTotalSeconds() != 0) {
387             prefix = prefix.concat(offset.getId());
388         }
389         return new ZoneRegion(prefix, offset.getRules());
390     }
391
392     /**
393      * Parses the ID, taking a flag to indicate whether {@code ZoneRulesException}
394      * should be thrown or not, used in deserialization.
395      *
396      * @param zoneId the time-zone ID, not null
397      * @param checkAvailable whether to check if the zone ID is available
398      * @return the zone ID, not null
399      * @throws DateTimeException if the ID format is invalid
400      * @throws ZoneRulesException if checking availability and the ID cannot be found
401      */
402     static ZoneId of(String zoneId, boolean checkAvailable) {
403         Objects.requireNonNull(zoneId, "zoneId");
404         if (zoneId.length() <= 1 || zoneId.startsWith("+") || zoneId.startsWith("-")) {
405             return ZoneOffset.of(zoneId);
406         } else if (zoneId.startsWith("UTC") || zoneId.startsWith("GMT")) {
407             return ofWithPrefix(zoneId, 3, checkAvailable);
408         } else if (zoneId.startsWith("UT")) {
409             return ofWithPrefix(zoneId, 2, checkAvailable);

```

```

410     }
411     return ZoneRegion.ofId(zoneId, checkAvailable);
412 }
413
414 /**
415  * Parse once a prefix is established.
416  *
417  * @param zoneId the time-zone ID, not null
418  * @param prefixLength the length of the prefix, 2 or 3
419  * @return the zone ID, not null
420  * @throws DateTimeException if the zone ID has an invalid format
421  */
422 private static ZoneId ofWithPrefix(String zoneId, int prefixLength, boolean checkAvailable) {
423     String prefix = zoneId.substring(0, prefixLength);
424     if (zoneId.length() == prefixLength) {
425         return ofOffset(prefix, ZoneOffset.UTC);
426     }
427     if (zoneId.charAt(prefixLength) != '+' && zoneId.charAt(prefixLength) != '-') {
428         return ZoneRegion.ofId(zoneId, checkAvailable); // drop through to ZoneRulesProvider
429     }
430     try {
431         ZoneOffset offset = ZoneOffset.of(zoneId.substring(prefixLength));
432         if (offset == ZoneOffset.UTC) {
433             return ofOffset(prefix, offset);
434         }
435         return ofOffset(prefix, offset);
436     } catch (DateTimeException ex) {
437         throw new DateTimeException("Invalid ID for offset-based ZoneId: " + zoneId, ex);
438     }
439 }
440
441 //-----
442 /**
443  * Obtains an instance of {@code ZoneId} from a temporal object.
444  * <p>
445  * This obtains a zone based on the specified temporal.
446  * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
447  * which this factory converts to an instance of {@code ZoneId}.
448  * <p>
449  * A {@code TemporalAccessor} represents some form of date and time information.
450  * This factory converts the arbitrary temporal object to an instance of {@code ZoneId}.
451  * <p>
452  * The conversion will try to obtain the zone in a way that favours region-based
453  * zones over offset-based zones using {@link TemporalQueries#zone()}.
454  * <p>
455  * This method matches the signature of the functional interface {@link TemporalQuery}
456  * allowing it to be used as a query via method reference, {@code ZoneId::from}.
457  *
458  * @param temporal the temporal object to convert, not null
459  * @return the zone ID, not null
460  * @throws DateTimeException if unable to convert to a {@code ZoneId}
461  */
462 public static ZoneId from(TemporalAccessor temporal) {

```

```

463     ZoneId obj = temporal.query(TemporalQueries.zone());
464     if (obj == null) {
465         throw new DateTimeException("Unable to obtain ZoneId from TemporalAccessor: " +
466             temporal + " of type " + temporal.getClass().getName());
467     }
468     return obj;
469 }
470
471 //-----
472 /**
473  * Constructor only accessible within the package.
474  */
475 ZoneId() {
476     if (getClass() != ZoneOffset.class && getClass() != ZoneRegion.class) {
477         throw new AssertionError("Invalid subclass");
478     }
479 }
480
481 //-----
482 /**
483  * Gets the unique time-zone ID.
484  * <p>
485  * This ID uniquely defines this object.
486  * The format of an offset based ID is defined by {@link ZoneOffset#getId()}.
487  *
488  * @return the time-zone unique ID, not null
489  */
490 public abstract String getId();
491
492 //-----
493 /**
494  * Gets the textual representation of the zone, such as 'British Time' or
495  * '+02:00'.
496  * <p>
497  * This returns the textual name used to identify the time-zone ID,
498  * suitable for presentation to the user.
499  * The parameters control the style of the returned text and the locale.
500  * <p>
501  * If no textual mapping is found then the {@link #getId() full ID} is returned.
502  *
503  * @param style the length of the text required, not null
504  * @param locale the locale to use, not null
505  * @return the text value of the zone, not null
506  */
507 public String getDisplayName(TextStyle style, Locale locale) {
508     return new DateTimeFormatterBuilder().appendZoneText(style).toFormatter(locale).format(
509         toTemporal());
510 }
511
512 /**
513  * Converts this zone to a {@code TemporalAccessor}.
514  * <p>
515  * A {@code ZoneId} can be fully represented as a {@code TemporalAccessor}.

```

```

515     * However, the interface is not implemented by this class as most of the
516     * methods on the interface have no meaning to {@code ZoneId}.
517     * <p>
518     * The returned temporal has no supported fields, with the query method
519     * supporting the return of the zone using {@link TemporalQueries#zoneId()}.
520     *
521     * @return a temporal equivalent to this zone, not null
522     */
523     private TemporalAccessor toTemporal() {
524         return new TemporalAccessor() {
525             @Override
526             public boolean isSupported(TemporalField field) {
527                 return false;
528             }
529             @Override
530             public long getLong(TemporalField field) {
531                 throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
532             }
533             @SuppressWarnings("unchecked")
534             @Override
535             public <R> R query(TemporalQuery<R> query) {
536                 if (query == TemporalQueries.zoneId()) {
537                     return (R) ZoneId.this;
538                 }
539                 return TemporalAccessor.super.query(query);
540             }
541         };
542     }
543
544     //-----
545     /**
546     * Gets the time-zone rules for this ID allowing calculations to be performed.
547     * <p>
548     * The rules provide the functionality associated with a time-zone,
549     * such as finding the offset for a given instant or local date-time.
550     * <p>
551     * A time-zone can be invalid if it is deserialized in a Java Runtime which
552     * does not have the same rules loaded as the Java Runtime that stored it.
553     * In this case, calling this method will throw a {@code ZoneRulesException}.
554     * <p>
555     * The rules are supplied by {@link ZoneRulesProvider}. An advanced provider may
556     * support dynamic updates to the rules without restarting the Java Runtime.
557     * If so, then the result of this method may change over time.
558     * Each individual call will be still remain thread-safe.
559     * <p>
560     * {@link ZoneOffset} will always return a set of rules where the offset never changes.
561     *
562     * @return the rules, not null
563     * @throws ZoneRulesException if no rules are available for this ID
564     */
565     public abstract ZoneRules getRules();
566
567     /**

```

```

568     * Normalizes the time-zone ID, returning a {@code ZoneOffset} where possible.
569     * <p>
570     * The returns a normalized {@code ZoneId} that can be used in place of this ID.
571     * The result will have {@code ZoneRules} equivalent to those returned by this object,
572     * however the ID returned by {@code getId()} may be different.
573     * <p>
574     * The normalization checks if the rules of this {@code ZoneId} have a fixed offset.
575     * If they do, then the {@code ZoneOffset} equal to that offset is returned.
576     * Otherwise {@code this} is returned.
577     *
578     * @return the time-zone unique ID, not null
579     */
580     public ZoneId normalized() {
581         try {
582             ZoneRules rules = getRules();
583             if (rules.isFixedOffset()) {
584                 return rules.getOffset(Instant.EPOCH);
585             }
586         } catch (ZoneRulesException ex) {
587             // invalid ZoneRegion is not important to this method
588         }
589         return this;
590     }
591
592     //-----
593     /**
594     * Checks if this time-zone ID is equal to another time-zone ID.
595     * <p>
596     * The comparison is based on the ID.
597     *
598     * @param obj the object to check, null returns false
599     * @return true if this is equal to the other time-zone ID
600     */
601     @Override
602     public boolean equals(Object obj) {
603         if (this == obj) {
604             return true;
605         }
606         if (obj instanceof ZoneId) {
607             ZoneId other = (ZoneId) obj;
608             return getId().equals(other.getId());
609         }
610         return false;
611     }
612
613     /**
614     * A hash code for this time-zone ID.
615     *
616     * @return a suitable hash code
617     */
618     @Override
619     public int hashCode() {
620         return getId().hashCode();

```

```

621     }
622
623     //-----
624     /**
625      * Defend against malicious streams.
626      *
627      * @param s the stream to read
628      * @throws InvalidObjectException always
629      */
630     private void readObject(ObjectInputStream s) throws InvalidObjectException {
631         throw new InvalidObjectException("Deserialization via serialization delegate");
632     }
633
634     /**
635      * Outputs this zone as a {@code String}, using the ID.
636      *
637      * @return a string representation of this time-zone ID, not null
638      */
639     @Override
640     public String toString() {
641         return getId();
642     }
643
644     //-----
645     /**
646      * Writes the object using a
647      * <a href="../../serialized-form.html#java.time.Ser">dedicated serialized form</a>.
648      * @serialData
649      * <pre>
650      *   out.writeByte(7); // identifies a ZoneId (not ZoneOffset)
651      *   out.writeUTF(getId());
652      * </pre>
653      * <p>
654      * When read back in, the {@code ZoneId} will be created as though using
655      * {@link #of(String)}, but without any exception in the case where the
656      * ID has a valid format, but is not in the known set of region-based IDs.
657      *
658      * @return the instance of {@code Ser}, not null
659      */
660     // this is here for serialization Javadoc
661     private Object writeReplace() {
662         return new Ser(Ser.ZONE_REGION_TYPE, this);
663     }
664
665     abstract void write(DataOutput out) throws IOException;
666
667 }

```

4.18 ZoneOffset.java

Secuencia 67: java/time/ZoneOffset.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.

```

```
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
```



```
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time;
63
64  import static java.time.LocalTime.MINUTES_PER_HOUR;
65  import static java.time.LocalTime.SECONDS_PER_HOUR;
66  import static java.time.LocalTime.SECONDS_PER_MINUTE;
67  import static java.time.temporal.ChronoField.OFFSET_SECONDS;
68
69  import java.io.DataInput;
70  import java.io.DataOutput;
71  import java.io.IOException;
72  import java.io.InvalidObjectException;
73  import java.io.ObjectInputStream;
74  import java.io.Serializable;
75  import java.time.temporal.ChronoField;
76  import java.time.temporal.Temporal;
77  import java.time.temporal.TemporalAccessor;
78  import java.time.temporal.TemporalAdjuster;
79  import java.time.temporal.TemporalField;
80  import java.time.temporal.TemporalQueries;
81  import java.time.temporal.TemporalQuery;
82  import java.time.temporal.UnsupportedTemporalTypeException;
83  import java.time.temporal.ValueRange;
84  import java.time.zone.ZoneRules;
85  import java.util.Objects;
86  import java.util.concurrent.ConcurrentHashMap;
87  import java.util.concurrent.ConcurrentMap;
88
89  /**
90   * A time-zone offset from Greenwich/UTC, such as {@code +02:00}.
91   * <p>
92   * A time-zone offset is the amount of time that a time-zone differs from Greenwich/UTC.
93   * This is usually a fixed number of hours and minutes.
94   * <p>
95   * Different parts of the world have different time-zone offsets.
96   * The rules for how offsets vary by place and time of year are captured in the
97   * {@link ZoneId} class.
98   * <p>
99   * For example, Paris is one hour ahead of Greenwich/UTC in winter and two hours
100  * ahead in summer. The {@code ZoneId} instance for Paris will reference two
101  * {@code ZoneOffset} instances - a {@code +01:00} instance for winter,
102  * and a {@code +02:00} instance for summer.
103  * <p>
104  * In 2008, time-zone offsets around the world extended from -12:00 to +14:00.
105  * To prevent any problems with that range being extended, yet still provide
106  * validation, the range of offsets is restricted to -18:00 to 18:00 inclusive.
107  * <p>
108  * This class is designed for use with the ISO calendar system.
```

```

109  * The fields of hours, minutes and seconds make assumptions that are valid for the
110  * standard ISO definitions of those fields. This class may be used with other
111  * calendar systems providing the definition of the time fields matches those
112  * of the ISO calendar system.
113  * <p>
114  * Instances of {@code ZoneOffset} must be compared using {@link #equals}.
115  * Implementations may choose to cache certain common offsets, however
116  * applications must not rely on such caching.
117  *
118  * <p>
119  * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
120  * class; use of identity-sensitive operations (including reference equality
121  * ({@code ==}), identity hash code, or synchronization) on instances of
122  * {@code ZoneOffset} may have unpredictable results and should be avoided.
123  * The {@code equals} method should be used for comparisons.
124  *
125  * @implSpec
126  * This class is immutable and thread-safe.
127  *
128  * @since 1.8
129  */
130 public final class ZoneOffset
131     extends ZoneId
132     implements TemporalAccessor, TemporalAdjuster, Comparable<ZoneOffset>, Serializable {
133
134     /** Cache of time-zone offset by offset in seconds. */
135     private static final ConcurrentMap<Integer, ZoneOffset> SECONDS_CACHE = new ConcurrentHashMap<>
        (16, 0.75f, 4);
136
137     /** Cache of time-zone offset by ID. */
138     private static final ConcurrentMap<String, ZoneOffset> ID_CACHE = new ConcurrentHashMap<>(16,
        0.75f, 4);
139
140     /**
141      * The abs maximum seconds.
142      */
143     private static final int MAX_SECONDS = 18 * SECONDS_PER_HOUR;
144
145     /**
146      * Serialization version.
147      */
148     private static final long serialVersionUID = 2357656521762053153L;
149
150     /**
151      * The time-zone offset for UTC, with an ID of 'Z'.
152      */
153     public static final ZoneOffset UTC = ZoneOffset.ofTotalSeconds(0);
154
155     /**
156      * Constant for the maximum supported offset.
157      */
158     public static final ZoneOffset MIN = ZoneOffset.ofTotalSeconds(-MAX_SECONDS);
159
160     /**
161      * Constant for the maximum supported offset.
162      */
163     public static final ZoneOffset MAX = ZoneOffset.ofTotalSeconds(MAX_SECONDS);

```

```

160
161 /**
162  * The total offset in seconds.
163  */
164 private final int totalSeconds;
165 /**
166  * The string form of the time-zone offset.
167  */
168 private final transient String id;
169
170 //-----
171 /**
172  * Obtains an instance of {@code ZoneOffset} using the ID.
173  * <p>
174  * This method parses the string ID of a {@code ZoneOffset} to
175  * return an instance. The parsing accepts all the formats generated by
176  * {@link #getId()}, plus some additional formats:
177  * <ul>
178  * <li>{@code Z} - for UTC
179  * <li>{@code +h}
180  * <li>{@code +hh}
181  * <li>{@code +hh:mm}
182  * <li>{@code -hh:mm}
183  * <li>{@code +hhmm}
184  * <li>{@code -hhmm}
185  * <li>{@code +hh:mm:ss}
186  * <li>{@code -hh:mm:ss}
187  * <li>{@code +hhmmss}
188  * <li>{@code -hhmmss}
189  * </ul>
190  * Note that &plusmn; means either the plus or minus symbol.
191  * <p>
192  * The ID of the returned offset will be normalized to one of the formats
193  * described by {@link #getId()}.
194  * <p>
195  * The maximum supported range is from +18:00 to -18:00 inclusive.
196  *
197  * @param offsetId the offset ID, not null
198  * @return the zone-offset, not null
199  * @throws DateTimeException if the offset ID is invalid
200  */
201 @SuppressWarnings("fallthrough")
202 public static ZoneOffset of(String offsetId) {
203     Objects.requireNonNull(offsetId, "offsetId");
204     // "Z" is always in the cache
205     ZoneOffset offset = ID_CACHE.get(offsetId);
206     if (offset != null) {
207         return offset;
208     }
209
210     // parse - +h, +hh, +hhmm, +hh:mm, +hhmmss, +hh:mm:ss
211     final int hours, minutes, seconds;
212     switch (offsetId.length()) {

```

```

213         case 2:
214             offsetId = offsetId.charAt(0) + "0" + offsetId.charAt(1); // fallthru
215         case 3:
216             hours = parseNumber(offsetId, 1, false);
217             minutes = 0;
218             seconds = 0;
219             break;
220         case 5:
221             hours = parseNumber(offsetId, 1, false);
222             minutes = parseNumber(offsetId, 3, false);
223             seconds = 0;
224             break;
225         case 6:
226             hours = parseNumber(offsetId, 1, false);
227             minutes = parseNumber(offsetId, 4, true);
228             seconds = 0;
229             break;
230         case 7:
231             hours = parseNumber(offsetId, 1, false);
232             minutes = parseNumber(offsetId, 3, false);
233             seconds = parseNumber(offsetId, 5, false);
234             break;
235         case 9:
236             hours = parseNumber(offsetId, 1, false);
237             minutes = parseNumber(offsetId, 4, true);
238             seconds = parseNumber(offsetId, 7, true);
239             break;
240         default:
241             throw new DateTimeException("Invalid ID for ZoneOffset, invalid format: " +
242                                     offsetId);
243     }
244     char first = offsetId.charAt(0);
245     if (first != '+' && first != '-') {
246         throw new DateTimeException("Invalid ID for ZoneOffset, plus/minus not found when
247                                     expected: " + offsetId);
248     }
249     if (first == '-') {
250         return ofHoursMinutesSeconds(-hours, -minutes, -seconds);
251     } else {
252         return ofHoursMinutesSeconds(hours, minutes, seconds);
253     }
254 }
255
256 /**
257  * Parse a two digit zero-prefixed number.
258  *
259  * @param offsetId the offset ID, not null
260  * @param pos the position to parse, valid
261  * @param precededByColon should this number be prefixed by a precededByColon
262  * @return the parsed number, from 0 to 99
263  */
264 private static int parseNumber(CharSequence offsetId, int pos, boolean precededByColon) {
265     if (precededByColon && offsetId.charAt(pos - 1) != ':') {

```

```

264         throw new DateTimeException("Invalid ID for ZoneOffset, colon not found when expected:
           " + offsetId);
265     }
266     char ch1 = offsetId.charAt(pos);
267     char ch2 = offsetId.charAt(pos + 1);
268     if (ch1 < '0' || ch1 > '9' || ch2 < '0' || ch2 > '9') {
269         throw new DateTimeException("Invalid ID for ZoneOffset, non numeric characters found: "
           + offsetId);
270     }
271     return (ch1 - 48) * 10 + (ch2 - 48);
272 }
273
274 //-----
275 /**
276  * Obtains an instance of {@code ZoneOffset} using an offset in hours.
277  *
278  * @param hours the time-zone offset in hours, from -18 to +18
279  * @return the zone-offset, not null
280  * @throws DateTimeException if the offset is not in the required range
281  */
282 public static ZoneOffset ofHours(int hours) {
283     return ofHoursMinutesSeconds(hours, 0, 0);
284 }
285
286 /**
287  * Obtains an instance of {@code ZoneOffset} using an offset in
288  * hours and minutes.
289  * <p>
290  * The sign of the hours and minutes components must match.
291  * Thus, if the hours is negative, the minutes must be negative or zero.
292  * If the hours is zero, the minutes may be positive, negative or zero.
293  *
294  * @param hours the time-zone offset in hours, from -18 to +18
295  * @param minutes the time-zone offset in minutes, from 0 to &plusmn;59, sign matches hours
296  * @return the zone-offset, not null
297  * @throws DateTimeException if the offset is not in the required range
298  */
299 public static ZoneOffset ofHoursMinutes(int hours, int minutes) {
300     return ofHoursMinutesSeconds(hours, minutes, 0);
301 }
302
303 /**
304  * Obtains an instance of {@code ZoneOffset} using an offset in
305  * hours, minutes and seconds.
306  * <p>
307  * The sign of the hours, minutes and seconds components must match.
308  * Thus, if the hours is negative, the minutes and seconds must be negative or zero.
309  *
310  * @param hours the time-zone offset in hours, from -18 to +18
311  * @param minutes the time-zone offset in minutes, from 0 to &plusmn;59, sign matches hours
           and seconds
312  * @param seconds the time-zone offset in seconds, from 0 to &plusmn;59, sign matches hours
           and minutes

```

```

313     * @return the zone-offset, not null
314     * @throws DateTimeException if the offset is not in the required range
315     */
316     public static ZoneOffset ofHoursMinutesSeconds(int hours, int minutes, int seconds) {
317         validate(hours, minutes, seconds);
318         int totalSeconds = totalSeconds(hours, minutes, seconds);
319         return ofTotalSeconds(totalSeconds);
320     }
321
322     //-----
323     /**
324      * Obtains an instance of {@code ZoneOffset} from a temporal object.
325      * <p>
326      * This obtains an offset based on the specified temporal.
327      * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
328      * which this factory converts to an instance of {@code ZoneOffset}.
329      * <p>
330      * A {@code TemporalAccessor} represents some form of date and time information.
331      * This factory converts the arbitrary temporal object to an instance of {@code ZoneOffset}.
332      * <p>
333      * The conversion uses the {@link TemporalQueries#offset()} query, which relies
334      * on extracting the {@link ChronoField#OFFSET_SECONDS OFFSET_SECONDS} field.
335      * <p>
336      * This method matches the signature of the functional interface {@link TemporalQuery}
337      * allowing it to be used as a query via method reference, {@code ZoneOffset::from}.
338      *
339      * @param temporal the temporal object to convert, not null
340      * @return the zone-offset, not null
341      * @throws DateTimeException if unable to convert to an {@code ZoneOffset}
342      */
343     public static ZoneOffset from(TemporalAccessor temporal) {
344         Objects.requireNonNull(temporal, "temporal");
345         ZoneOffset offset = temporal.query(TemporalQueries.offset());
346         if (offset == null) {
347             throw new DateTimeException("Unable to obtain ZoneOffset from TemporalAccessor: " +
348                 temporal + " of type " + temporal.getClass().getName());
349         }
350         return offset;
351     }
352
353     //-----
354     /**
355      * Validates the offset fields.
356      *
357      * @param hours the time-zone offset in hours, from -18 to +18
358      * @param minutes the time-zone offset in minutes, from 0 to &plusmn;59
359      * @param seconds the time-zone offset in seconds, from 0 to &plusmn;59
360      * @throws DateTimeException if the offset is not in the required range
361      */
362     private static void validate(int hours, int minutes, int seconds) {
363         if (hours < -18 || hours > 18) {
364             throw new DateTimeException("Zone offset hours not in valid range: value " + hours +
365                 " is not in the range -18 to 18");
366         }
367     }

```

```

366     }
367     if (hours > 0) {
368         if (minutes < 0 || seconds < 0) {
369             throw new DateTimeException("Zone offset minutes and seconds must be positive
370                                     because hours is positive");
371         }
372     } else if (hours < 0) {
373         if (minutes > 0 || seconds > 0) {
374             throw new DateTimeException("Zone offset minutes and seconds must be negative
375                                     because hours is negative");
376         }
377     } else if ((minutes > 0 && seconds < 0) || (minutes < 0 && seconds > 0)) {
378         throw new DateTimeException("Zone offset minutes and seconds must have the same sign");
379     }
380     if (Math.abs(minutes) > 59) {
381         throw new DateTimeException("Zone offset minutes not in valid range: abs(value) " +
382                                     Math.abs(minutes) + " is not in the range 0 to 59");
383     }
384     if (Math.abs(seconds) > 59) {
385         throw new DateTimeException("Zone offset seconds not in valid range: abs(value) " +
386                                     Math.abs(seconds) + " is not in the range 0 to 59");
387     }
388     if (Math.abs(hours) == 18 && (Math.abs(minutes) > 0 || Math.abs(seconds) > 0)) {
389         throw new DateTimeException("Zone offset not in valid range: -18:00 to +18:00");
390     }
391 }
392
393 /**
394  * Calculates the total offset in seconds.
395  *
396  * @param hours the time-zone offset in hours, from -18 to +18
397  * @param minutes the time-zone offset in minutes, from 0 to &plusmn;59, sign matches hours
398  *               and seconds
399  * @param seconds the time-zone offset in seconds, from 0 to &plusmn;59, sign matches hours
400  *               and minutes
401  * @return the total in seconds
402  */
403 private static int totalSeconds(int hours, int minutes, int seconds) {
404     return hours * SECONDS_PER_HOUR + minutes * SECONDS_PER_MINUTE + seconds;
405 }
406
407 //-----
408 /**
409  * Obtains an instance of {@code ZoneOffset} specifying the total offset in seconds
410  * <p>
411  * The offset must be in the range {@code -18:00} to {@code +18:00}, which corresponds to
412  * -64800 to +64800.
413  *
414  * @param totalSeconds the total time-zone offset in seconds, from -64800 to +64800
415  * @return the ZoneOffset, not null
416  * @throws DateTimeException if the offset is not in the required range
417  */
418 public static ZoneOffset ofTotalSeconds(int totalSeconds) {

```

```

414     if (Math.abs(totalSeconds) > MAX_SECONDS) {
415         throw new DateTimeException("Zone offset not in valid range: -18:00 to +18:00");
416     }
417     if (totalSeconds % (15 * SECONDS_PER_MINUTE) == 0) {
418         Integer totalSecs = totalSeconds;
419         ZoneOffset result = SECONDS_CACHE.get(totalSecs);
420         if (result == null) {
421             result = new ZoneOffset(totalSeconds);
422             SECONDS_CACHE.putIfAbsent(totalSecs, result);
423             result = SECONDS_CACHE.get(totalSecs);
424             ID_CACHE.putIfAbsent(result.getId(), result);
425         }
426         return result;
427     } else {
428         return new ZoneOffset(totalSeconds);
429     }
430 }
431
432 //-----
433 /**
434  * Constructor.
435  *
436  * @param totalSeconds the total time-zone offset in seconds, from -64800 to +64800
437  */
438 private ZoneOffset(int totalSeconds) {
439     super();
440     this.totalSeconds = totalSeconds;
441     id = buildId(totalSeconds);
442 }
443
444 private static String buildId(int totalSeconds) {
445     if (totalSeconds == 0) {
446         return "Z";
447     } else {
448         int absTotalSeconds = Math.abs(totalSeconds);
449         StringBuilder buf = new StringBuilder();
450         int absHours = absTotalSeconds / SECONDS_PER_HOUR;
451         int absMinutes = (absTotalSeconds / SECONDS_PER_MINUTE) % MINUTES_PER_HOUR;
452         buf.append(totalSeconds < 0 ? "-" : "+")
453             .append(absHours < 10 ? "0" : "").append(absHours)
454             .append(absMinutes < 10 ? ":0" : ":").append(absMinutes);
455         int absSeconds = absTotalSeconds % SECONDS_PER_MINUTE;
456         if (absSeconds != 0) {
457             buf.append(absSeconds < 10 ? ":0" : ":").append(absSeconds);
458         }
459         return buf.toString();
460     }
461 }
462
463 //-----
464 /**
465  * Gets the total zone offset in seconds.
466  * <p>

```



```

467     * This is the primary way to access the offset amount.
468     * It returns the total of the hours, minutes and seconds fields as a
469     * single offset that can be added to a time.
470     *
471     * @return the total zone offset amount in seconds
472     */
473     public int getTotalSeconds() {
474         return totalSeconds;
475     }
476
477     /**
478     * Gets the normalized zone offset ID.
479     * <p>
480     * The ID is minor variation to the standard ISO-8601 formatted string
481     * for the offset. There are three formats:
482     * <ul>
483     * <li>{@code Z} - for UTC (ISO-8601)
484     * <li>{@code +hh:mm} or {@code -hh:mm} - if the seconds are zero (ISO-8601)
485     * <li>{@code +hh:mm:ss} or {@code -hh:mm:ss} - if the seconds are non-zero (not ISO-8601)
486     * </ul>
487     *
488     * @return the zone offset ID, not null
489     */
490     @Override
491     public String getId() {
492         return id;
493     }
494
495     /**
496     * Gets the associated time-zone rules.
497     * <p>
498     * The rules will always return this offset when queried.
499     * The implementation class is immutable, thread-safe and serializable.
500     *
501     * @return the rules, not null
502     */
503     @Override
504     public ZoneRules getRules() {
505         return ZoneRules.of(this);
506     }
507
508     //-----
509     /**
510     * Checks if the specified field is supported.
511     * <p>
512     * This checks if this offset can be queried for the specified field.
513     * If false, then calling the {@link #range(TemporalField) range} and
514     * {@link #get(TemporalField) get} methods will throw an exception.
515     * <p>
516     * If the field is a {@link ChronoField} then the query is implemented here.
517     * The {@code OFFSET_SECONDS} field returns true.
518     * All other {@code ChronoField} instances will return false.
519     * <p>

```

```

520     * If the field is not a {@code ChronoField}, then the result of this method
521     * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
522     * passing {@code this} as the argument.
523     * Whether the field is supported is determined by the field.
524     *
525     * @param field the field to check, null returns false
526     * @return true if the field is supported on this offset, false if not
527     */
528     @Override
529     public boolean isSupported(TemporalField field) {
530         if (field instanceof ChronoField) {
531             return field == OFFSET_SECONDS;
532         }
533         return field != null && field.isSupportedBy(this);
534     }
535
536     /**
537     * Gets the range of valid values for the specified field.
538     * <p>
539     * The range object expresses the minimum and maximum valid values for a field.
540     * This offset is used to enhance the accuracy of the returned range.
541     * If it is not possible to return the range, because the field is not supported
542     * or for some other reason, an exception is thrown.
543     * <p>
544     * If the field is a {@link ChronoField} then the query is implemented here.
545     * The {@link #isSupported(TemporalField) supported fields} will return
546     * appropriate range instances.
547     * All other {@code ChronoField} instances will throw an {@code
548         UnsupportedOperationException}.
549     * <p>
550     * If the field is not a {@code ChronoField}, then the result of this method
551     * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
552     * passing {@code this} as the argument.
553     * Whether the range can be obtained is determined by the field.
554     *
555     * @param field the field to query the range for, not null
556     * @return the range of valid values for the field, not null
557     * @throws DateTimeException if the range for the field cannot be obtained
558     * @throws UnsupportedOperationException if the field is not supported
559     */
560     @Override // override for Javadoc
561     public ValueRange range(TemporalField field) {
562         return TemporalAccessor.super.range(field);
563     }
564
565     /**
566     * Gets the value of the specified field from this offset as an {@code int}.
567     * <p>
568     * This queries this offset for the value of the specified field.
569     * The returned value will always be within the valid range of values for the field.
570     * If it is not possible to return the value, because the field is not supported
571     * or for some other reason, an exception is thrown.
572     * <p>

```

```

572     * If the field is a {@link ChronoField} then the query is implemented here.
573     * The {@code OFFSET_SECONDS} field returns the value of the offset.
574     * All other {@code ChronoField} instances will throw an {@code
        UnsupportedTemporalTypeException}.
575     * <p>
576     * If the field is not a {@code ChronoField}, then the result of this method
577     * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
578     * passing {@code this} as the argument. Whether the value can be obtained,
579     * and what the value represents, is determined by the field.
580     *
581     * @param field the field to get, not null
582     * @return the value for the field
583     * @throws DateTimeException if a value for the field cannot be obtained or
584     *         the value is outside the range of valid values for the field
585     * @throws UnsupportedTemporalTypeException if the field is not supported or
586     *         the range of values exceeds an {@code int}
587     * @throws ArithmeticException if numeric overflow occurs
588     */
589     @Override // override for Javadoc and performance
590     public int get(TemporalField field) {
591         if (field == OFFSET_SECONDS) {
592             return totalSeconds;
593         } else if (field instanceof ChronoField) {
594             throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
595         }
596         return range(field).checkValidIntValue(getLong(field), field);
597     }
598
599     /**
600     * Gets the value of the specified field from this offset as a {@code long}.
601     * <p>
602     * This queries this offset for the value of the specified field.
603     * If it is not possible to return the value, because the field is not supported
604     * or for some other reason, an exception is thrown.
605     * <p>
606     * If the field is a {@link ChronoField} then the query is implemented here.
607     * The {@code OFFSET_SECONDS} field returns the value of the offset.
608     * All other {@code ChronoField} instances will throw an {@code
        UnsupportedTemporalTypeException}.
609     * <p>
610     * If the field is not a {@code ChronoField}, then the result of this method
611     * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
612     * passing {@code this} as the argument. Whether the value can be obtained,
613     * and what the value represents, is determined by the field.
614     *
615     * @param field the field to get, not null
616     * @return the value for the field
617     * @throws DateTimeException if a value for the field cannot be obtained
618     * @throws UnsupportedTemporalTypeException if the field is not supported
619     * @throws ArithmeticException if numeric overflow occurs
620     */
621     @Override
622     public long getLong(TemporalField field) {

```

```

623     if (field == OFFSET_SECONDS) {
624         return totalSeconds;
625     } else if (field instanceof ChronoField) {
626         throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
627     }
628     return field.getFrom(this);
629 }
630
631 //-----
632 /**
633  * Queries this offset using the specified query.
634  * <p>
635  * This queries this offset using the specified query strategy object.
636  * The {@code TemporalQuery} object defines the logic to be used to
637  * obtain the result. Read the documentation of the query to understand
638  * what the result of this method will be.
639  * <p>
640  * The result of this method is obtained by invoking the
641  * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
642  * specified query passing {@code this} as the argument.
643  *
644  * @param <R> the type of the result
645  * @param query the query to invoke, not null
646  * @return the query result, null may be returned (defined by the query)
647  * @throws DateTimeException if unable to query (defined by the query)
648  * @throws ArithmeticException if numeric overflow occurs (defined by the query)
649  */
650 @SuppressWarnings("unchecked")
651 @Override
652 public <R> R query(TemporalQuery<R> query) {
653     if (query == TemporalQueries.offset() || query == TemporalQueries.zone()) {
654         return (R) this;
655     }
656     return TemporalAccessor.super.query(query);
657 }
658
659 /**
660  * Adjusts the specified temporal object to have the same offset as this object.
661  * <p>
662  * This returns a temporal object of the same observable type as the input
663  * with the offset changed to be the same as this.
664  * <p>
665  * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
666  * passing {@link ChronoField#OFFSET_SECONDS} as the field.
667  * <p>
668  * In most cases, it is clearer to reverse the calling pattern by using
669  * {@link Temporal#with(TemporalAdjuster)}:
670  * <pre>
671  * // these two lines are equivalent, but the second approach is recommended
672  * temporal = thisOffset.adjustInto(temporal);
673  * temporal = temporal.with(thisOffset);
674  * </pre>
675  * <p>

```

```

676     * This instance is immutable and unaffected by this method call.
677     *
678     * @param temporal the target object to be adjusted, not null
679     * @return the adjusted object, not null
680     * @throws DateTimeException if unable to make the adjustment
681     * @throws ArithmeticException if numeric overflow occurs
682     */
683     @Override
684     public Temporal adjustInto(Temporal temporal) {
685         return temporal.with(OFFSET_SECONDS, totalSeconds);
686     }
687
688     //-----
689     /**
690     * Compares this offset to another offset in descending order.
691     * <p>
692     * The offsets are compared in the order that they occur for the same time
693     * of day around the world. Thus, an offset of {@code +10:00} comes before an
694     * offset of {@code +09:00} and so on down to {@code -18:00}.
695     * <p>
696     * The comparison is "consistent with equals", as defined by {@link Comparable}.
697     *
698     * @param other the other date to compare to, not null
699     * @return the comparator value, negative if less, positive if greater
700     * @throws NullPointerException if {@code other} is null
701     */
702     @Override
703     public int compareTo(ZoneOffset other) {
704         return other.totalSeconds - totalSeconds;
705     }
706
707     //-----
708     /**
709     * Checks if this offset is equal to another offset.
710     * <p>
711     * The comparison is based on the amount of the offset in seconds.
712     * This is equivalent to a comparison by ID.
713     *
714     * @param obj the object to check, null returns false
715     * @return true if this is equal to the other offset
716     */
717     @Override
718     public boolean equals(Object obj) {
719         if (this == obj) {
720             return true;
721         }
722         if (obj instanceof ZoneOffset) {
723             return totalSeconds == ((ZoneOffset) obj).totalSeconds;
724         }
725         return false;
726     }
727
728     /**

```

```

729     * A hash code for this offset.
730     *
731     * @return a suitable hash code
732     */
733     @Override
734     public int hashCode() {
735         return totalSeconds;
736     }
737
738     //-----
739     /**
740     * Outputs this offset as a {@code String}, using the normalized ID.
741     *
742     * @return a string representation of this offset, not null
743     */
744     @Override
745     public String toString() {
746         return id;
747     }
748
749     // -----
750     /**
751     * Writes the object using a
752     * <a href=".../serialized-form.html#java.time.Ser">dedicated serialized form</a>.
753     * @serialData
754     * <pre>
755     *   out.writeByte(8);           // identifies a ZoneOffset
756     *   int offsetByte = totalSeconds % 900 == 0 ? totalSeconds / 900 : 127;
757     *   out.writeByte(offsetByte);
758     *   if (offsetByte == 127) {
759     *       out.writeInt(totalSeconds);
760     *   }
761     * </pre>
762     *
763     * @return the instance of {@code Ser}, not null
764     */
765     private Object writeReplace() {
766         return new Ser(Ser.ZONE_OFFSET_TYPE, this);
767     }
768
769     /**
770     * Defend against malicious streams.
771     *
772     * @param s the stream to read
773     * @throws InvalidObjectException always
774     */
775     private void readObject(ObjectInputStream s) throws InvalidObjectException {
776         throw new InvalidObjectException("Deserialization via serialization delegate");
777     }
778
779     @Override
780     void write(DataOutput out) throws IOException {
781         out.writeByte(Ser.ZONE_OFFSET_TYPE);

```

```

782     writeExternal(out);
783 }
784
785 void writeExternal(DataOutput out) throws IOException {
786     final int offsetSecs = totalSeconds;
787     int offsetByte = offsetSecs % 900 == 0 ? offsetSecs / 900 : 127; // compress to -72 to +72
788     out.writeByte(offsetByte);
789     if (offsetByte == 127) {
790         out.writeInt(offsetSecs);
791     }
792 }
793
794 static ZoneOffset readExternal(DataInput in) throws IOException {
795     int offsetByte = in.readByte();
796     return (offsetByte == 127 ? ZoneOffset.ofTotalSeconds(in.readInt()) : ZoneOffset.
        ofTotalSeconds(offsetByte * 900));
797 }
798
799 }

```

4.19 ZoneRegion.java

Secuencia 68: java/time/ZoneRegion.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
28 *
29 * All rights reserved.
30 *

```

```
31 * Redistribution and use in source and binary forms, with or without
32 * modification, are permitted provided that the following conditions are met:
33 *
34 * * Redistributions of source code must retain the above copyright notice,
35 *   this list of conditions and the following disclaimer.
36 *
37 * * Redistributions in binary form must reproduce the above copyright notice,
38 *   this list of conditions and the following disclaimer in the documentation
39 *   and/or other materials provided with the distribution.
40 *
41 * * Neither the name of JSR-310 nor the names of its contributors
42 *   may be used to endorse or promote products derived from this software
43 *   without specific prior written permission.
44 *
45 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56 */
57 package java.time;
58
59 import java.io.DataInput;
60 import java.io.DataOutput;
61 import java.io.IOException;
62 import java.io.InvalidObjectException;
63 import java.io.ObjectInputStream;
64 import java.io.Serializable;
65 import java.time.zone.ZoneRules;
66 import java.time.zone.ZoneRulesException;
67 import java.time.zone.ZoneRulesProvider;
68 import java.util.Objects;
69
70 /**
71  * A geographical region where the same time-zone rules apply.
72  * <p>
73  * Time-zone information is categorized as a set of rules defining when and
74  * how the offset from UTC/Greenwich changes. These rules are accessed using
75  * identifiers based on geographical regions, such as countries or states.
76  * The most common region classification is the Time Zone Database (TZDB),
77  * which defines regions such as 'Europe/Paris' and 'Asia/Tokyo'.
78  * <p>
79  * The region identifier, modeled by this class, is distinct from the
80  * underlying rules, modeled by {@link ZoneRules}.
81  * The rules are defined by governments and change frequently.
82  * By contrast, the region identifier is well-defined and long-lived.
83  * This separation also allows rules to be shared between regions if appropriate.
```



```

84  *
85  * @implSpec
86  * This class is immutable and thread-safe.
87  *
88  * @since 1.8
89  */
90  final class ZoneRegion extends ZoneId implements Serializable {
91
92      /**
93       * Serialization version.
94       */
95      private static final long serialVersionUID = 8386373296231747096L;
96      /**
97       * The time-zone ID, not null.
98       */
99      private final String id;
100     /**
101      * The time-zone rules, null if zone ID was loaded leniently.
102      */
103     private final transient ZoneRules rules;
104
105     /**
106      * Obtains an instance of {@code ZoneId} from an identifier.
107      *
108      * @param zoneId the time-zone ID, not null
109      * @param checkAvailable whether to check if the zone ID is available
110      * @return the zone ID, not null
111      * @throws DateTimeException if the ID format is invalid
112      * @throws ZoneRulesException if checking availability and the ID cannot be found
113      */
114     static ZoneRegion ofId(String zoneId, boolean checkAvailable) {
115         Objects.requireNonNull(zoneId, "zoneId");
116         checkName(zoneId);
117         ZoneRules rules = null;
118         try {
119             // always attempt load for better behavior after deserialization
120             rules = ZoneRulesProvider.getRules(zoneId, true);
121         } catch (ZoneRulesException ex) {
122             if (checkAvailable) {
123                 throw ex;
124             }
125         }
126         return new ZoneRegion(zoneId, rules);
127     }
128
129     /**
130      * Checks that the given string is a legal ZondId name.
131      *
132      * @param zoneId the time-zone ID, not null
133      * @throws DateTimeException if the ID format is invalid
134      */
135     private static void checkName(String zoneId) {
136         int n = zoneId.length();

```

```

137         if (n < 2) {
138             throw new DateTimeException("Invalid ID for region-based ZoneId, invalid format: " +
139                                     zoneId);
140         }
141         for (int i = 0; i < n; i++) {
142             char c = zoneId.charAt(i);
143             if (c >= 'a' && c <= 'z') continue;
144             if (c >= 'A' && c <= 'Z') continue;
145             if (c == '/' && i != 0) continue;
146             if (c >= '0' && c <= '9' && i != 0) continue;
147             if (c == '~' && i != 0) continue;
148             if (c == '.' && i != 0) continue;
149             if (c == '_' && i != 0) continue;
150             if (c == '+' && i != 0) continue;
151             if (c == '-' && i != 0) continue;
152             throw new DateTimeException("Invalid ID for region-based ZoneId, invalid format: " +
153                                     zoneId);
154         }
155     }
156
157     //-----
158     /**
159     * Constructor.
160     *
161     * @param id the time-zone ID, not null
162     * @param rules the rules, null for lazy lookup
163     */
164     ZoneRegion(String id, ZoneRules rules) {
165         this.id = id;
166         this.rules = rules;
167     }
168
169     //-----
170     @Override
171     public String getId() {
172         return id;
173     }
174
175     @Override
176     public ZoneRules getRules() {
177         // additional query for group provider when null allows for possibility
178         // that the provider was updated after the ZoneId was created
179         return (rules != null ? rules : ZoneRulesProvider.getRules(id, false));
180     }
181
182     //-----
183     /**
184     * Writes the object using a
185     * <a href=".../serialized-form.html#java.time.Ser">dedicated serialized form</a>.
186     * @serialData
187     * <pre>
188     * out.writeByte(7); // identifies a ZoneId (not ZoneOffset)
189     * out.writeUTF(zoneId);

```

```

188     * </pre>
189     *
190     * @return the instance of {@code Ser}, not null
191     */
192     private Object writeReplace() {
193         return new Ser(Ser.ZONE_REGION_TYPE, this);
194     }
195
196     /**
197     * Defend against malicious streams.
198     *
199     * @param s the stream to read
200     * @throws InvalidObjectException always
201     */
202     private void readObject(ObjectInputStream s) throws InvalidObjectException {
203         throw new InvalidObjectException("Deserialization via serialization delegate");
204     }
205
206     @Override
207     void write(DataOutput out) throws IOException {
208         out.writeByte(Ser.ZONE_REGION_TYPE);
209         writeExternal(out);
210     }
211
212     void writeExternal(DataOutput out) throws IOException {
213         out.writeUTF(id);
214     }
215
216     static ZoneId readExternal(DataInput in) throws IOException {
217         String id = in.readUTF();
218         return ZoneId.of(id, false);
219     }
220
221 }

```

4.20 ZonedDateTime.java

Secuencia 69: java/time/ZonedDateTime.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *

```

```
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time;
63
64  import static java.time.temporal.ChronoField.INSTANT_SECONDS;
65  import static java.time.temporal.ChronoField.NANO_OF_SECOND;
66  import static java.time.temporal.ChronoField.OFFSET_SECONDS;
67
68  import java.io.DataOutput;
```

```
69 import java.io.IOException;
70 import java.io.ObjectInput;
71 import java.io.InvalidObjectException;
72 import java.io.ObjectInputStream;
73 import java.io.Serializable;
74 import java.time.chrono.ChronoZonedDateTime;
75 import java.time.format.DateTimeFormatter;
76 import java.time.format.DateTimeParseException;
77 import java.time.temporal.ChronoField;
78 import java.time.temporal.ChronoUnit;
79 import java.time.temporal.Temporal;
80 import java.time.temporal.TemporalAccessor;
81 import java.time.temporal.TemporalAdjuster;
82 import java.time.temporal.TemporalAmount;
83 import java.time.temporal.TemporalField;
84 import java.time.temporal.TemporalQueries;
85 import java.time.temporal.TemporalQuery;
86 import java.time.temporal.TemporalUnit;
87 import java.time.temporal.UnsupportedTemporalTypeException;
88 import java.time.temporal.ValueRange;
89 import java.time.zone.ZoneOffsetTransition;
90 import java.time.zone.ZoneRules;
91 import java.util.List;
92 import java.util.Objects;
93
94 /**
95  * A date-time with a time-zone in the ISO-8601 calendar system,
96  * such as {@code 2007-12-03T10:15:30+01:00 Europe/Paris}.
97  * <p>
98  * {@code ZonedDateTime} is an immutable representation of a date-time with a time-zone.
99  * This class stores all date and time fields, to a precision of nanoseconds,
100  * and a time-zone, with a zone offset used to handle ambiguous local date-times.
101  * For example, the value
102  * "2nd October 2007 at 13:45.30.123456789 +02:00 in the Europe/Paris time-zone"
103  * can be stored in a {@code ZonedDateTime}.
104  * <p>
105  * This class handles conversion from the local time-line of {@code LocalDateTime}
106  * to the instant time-line of {@code Instant}.
107  * The difference between the two time-lines is the offset from UTC/Greenwich,
108  * represented by a {@code ZoneOffset}.
109  * <p>
110  * Converting between the two time-lines involves calculating the offset using the
111  * {@link ZoneRules rules} accessed from the {@code ZoneId}.
112  * Obtaining the offset for an instant is simple, as there is exactly one valid
113  * offset for each instant. By contrast, obtaining the offset for a local date-time
114  * is not straightforward. There are three cases:
115  * <ul>
116  * <li>Normal, with one valid offset. For the vast majority of the year, the normal
117  * case applies, where there is a single valid offset for the local date-time.</li>
118  * <li>Gap, with zero valid offsets. This is when clocks jump forward typically
119  * due to the spring daylight savings change from "winter" to "summer".
120  * In a gap there are local date-time values with no valid offset.</li>
121  * <li>Overlap, with two valid offsets. This is when clocks are set back typically
```

```

122 * due to the autumn daylight savings change from "summer" to "winter".
123 * In an overlap there are local date-time values with two valid offsets.</li>
124 * </ul>
125 * <p>
126 * Any method that converts directly or implicitly from a local date-time to an
127 * instant by obtaining the offset has the potential to be complicated.
128 * <p>
129 * For Gaps, the general strategy is that if the local date-time falls in the
130 * middle of a Gap, then the resulting zoned date-time will have a local date-time
131 * shifted forwards by the length of the Gap, resulting in a date-time in the later
132 * offset, typically "summer" time.
133 * <p>
134 * For Overlaps, the general strategy is that if the local date-time falls in the
135 * middle of an Overlap, then the previous offset will be retained. If there is no
136 * previous offset, or the previous offset is invalid, then the earlier offset is
137 * used, typically "summer" time.. Two additional methods,
138 * {@link #withEarlierOffsetAtOverlap()} and {@link #withLaterOffsetAtOverlap()},
139 * help manage the case of an overlap.
140 * <p>
141 * In terms of design, this class should be viewed primarily as the combination
142 * of a {@code LocalDateTime} and a {@code ZoneId}. The {@code ZoneOffset} is
143 * a vital, but secondary, piece of information, used to ensure that the class
144 * represents an instant, especially during a daylight savings overlap.
145 *
146 * <p>
147 * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
148 * class; use of identity-sensitive operations (including reference equality
149 * ({@code ==}), identity hash code, or synchronization) on instances of
150 * {@code ZonedDateTime} may have unpredictable results and should be avoided.
151 * The {@code equals} method should be used for comparisons.
152 *
153 * @implSpec
154 * A {@code ZonedDateTime} holds state equivalent to three separate objects,
155 * a {@code LocalDateTime}, a {@code ZoneId} and the resolved {@code ZoneOffset}.
156 * The offset and local date-time are used to define an instant when necessary.
157 * The zone ID is used to obtain the rules for how and when the offset changes.
158 * The offset cannot be freely set, as the zone controls which offsets are valid.
159 * <p>
160 * This class is immutable and thread-safe.
161 *
162 * @since 1.8
163 */
164 public final class ZonedDateTime
165     implements Temporal, ChronoZonedDateTime<LocalDate>, Serializable {
166
167     /**
168      * Serialization version.
169      */
170     private static final long serialVersionUID = -6260982410461394882L;
171
172     /**
173      * The local date-time.
174      */

```

```

175     private final LocalDateTime dateTime;
176     /**
177      * The offset from UTC/Greenwich.
178      */
179     private final ZoneOffset offset;
180     /**
181      * The time-zone.
182      */
183     private final ZoneId zone;
184
185     //-----
186     /**
187      * Obtains the current date-time from the system clock in the default time-zone.
188      * <p>
189      * This will query the {@link Clock#systemDefaultZone() system clock} in the default
190      * time-zone to obtain the current date-time.
191      * The zone and offset will be set based on the time-zone in the clock.
192      * <p>
193      * Using this method will prevent the ability to use an alternate clock for testing
194      * because the clock is hard-coded.
195      *
196      * @return the current date-time using the system clock, not null
197      */
198     public static ZonedDateTime now() {
199         return now(Clock.systemDefaultZone());
200     }
201
202     /**
203      * Obtains the current date-time from the system clock in the specified time-zone.
204      * <p>
205      * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current date-
206      * time.
207      * Specifying the time-zone avoids dependence on the default time-zone.
208      * The offset will be calculated from the specified time-zone.
209      * <p>
210      * Using this method will prevent the ability to use an alternate clock for testing
211      * because the clock is hard-coded.
212      *
213      * @param zone the zone ID to use, not null
214      * @return the current date-time using the system clock, not null
215      */
216     public static ZonedDateTime now(ZoneId zone) {
217         return now(Clock.system(zone));
218     }
219
220     /**
221      * Obtains the current date-time from the specified clock.
222      * <p>
223      * This will query the specified clock to obtain the current date-time.
224      * The zone and offset will be set based on the time-zone in the clock.
225      * <p>
226      * Using this method allows the use of an alternate clock for testing.
227      * The alternate clock may be introduced using {@link Clock dependency injection}.

```

```

227     *
228     * @param clock the clock to use, not null
229     * @return the current date-time, not null
230     */
231     public static ZonedDateTime now(Clock clock) {
232         Objects.requireNonNull(clock, "clock");
233         final Instant now = clock.instant(); // called once
234         return ofInstant(now, clock.getZone());
235     }
236
237     //-----
238     /**
239     * Obtains an instance of {@code ZonedDateTime} from a local date and time.
240     * <p>
241     * This creates a zoned date-time matching the input local date and time as closely as possible
242     *
243     * Time-zone rules, such as daylight savings, mean that not every local date-time
244     * is valid for the specified zone, thus the local date-time may be adjusted.
245     * <p>
246     * The local date time and first combined to form a local date-time.
247     * The local date-time is then resolved to a single instant on the time-line.
248     * This is achieved by finding a valid offset from UTC/Greenwich for the local
249     * date-time as defined by the {@link ZoneRules rules} of the zone ID.
250     * <p>
251     * In most cases, there is only one valid offset for a local date-time.
252     * In the case of an overlap, when clocks are set back, there are two valid offsets.
253     * This method uses the earlier offset typically corresponding to "summer".
254     * <p>
255     * In the case of a gap, when clocks jump forward, there is no valid offset.
256     * Instead, the local date-time is adjusted to be later by the length of the gap.
257     * For a typical one hour daylight savings change, the local date-time will be
258     * moved one hour later into the offset typically corresponding to "summer".
259     *
260     * @param date the local date, not null
261     * @param time the local time, not null
262     * @param zone the time-zone, not null
263     * @return the offset date-time, not null
264     */
265     public static ZonedDateTime of(LocalDate date, LocalTime time, ZoneId zone) {
266         return of(LocalDateTime.of(date, time), zone);
267     }
268
269     /**
270     * Obtains an instance of {@code ZonedDateTime} from a local date-time.
271     * <p>
272     * This creates a zoned date-time matching the input local date-time as closely as possible.
273     * Time-zone rules, such as daylight savings, mean that not every local date-time
274     * is valid for the specified zone, thus the local date-time may be adjusted.
275     * <p>
276     * The local date-time is resolved to a single instant on the time-line.
277     * This is achieved by finding a valid offset from UTC/Greenwich for the local
278     * date-time as defined by the {@link ZoneRules rules} of the zone ID.
279     * <p>

```



```

279      * In most cases, there is only one valid offset for a local date-time.
280      * In the case of an overlap, when clocks are set back, there are two valid offsets.
281      * This method uses the earlier offset typically corresponding to "summer".
282      * <p>
283      * In the case of a gap, when clocks jump forward, there is no valid offset.
284      * Instead, the local date-time is adjusted to be later by the length of the gap.
285      * For a typical one hour daylight savings change, the local date-time will be
286      * moved one hour later into the offset typically corresponding to "summer".
287      *
288      * @param localDateTime the local date-time, not null
289      * @param zone the time-zone, not null
290      * @return the zoned date-time, not null
291      */
292      public static ZonedDateTime of(LocalDateTime localDateTime, ZoneId zone) {
293          return ofLocal(localDateTime, zone, null);
294      }
295
296      /**
297       * Obtains an instance of {@code ZonedDateTime} from a year, month, day,
298       * hour, minute, second, nanosecond and time-zone.
299       * <p>
300       * This creates a zoned date-time matching the local date-time of the seven
301       * specified fields as closely as possible.
302       * Time-zone rules, such as daylight savings, mean that not every local date-time
303       * is valid for the specified zone, thus the local date-time may be adjusted.
304       * <p>
305       * The local date-time is resolved to a single instant on the time-line.
306       * This is achieved by finding a valid offset from UTC/Greenwich for the local
307       * date-time as defined by the {@link ZoneRules rules} of the zone ID.
308       * <p>
309       * In most cases, there is only one valid offset for a local date-time.
310       * In the case of an overlap, when clocks are set back, there are two valid offsets.
311       * This method uses the earlier offset typically corresponding to "summer".
312       * <p>
313       * In the case of a gap, when clocks jump forward, there is no valid offset.
314       * Instead, the local date-time is adjusted to be later by the length of the gap.
315       * For a typical one hour daylight savings change, the local date-time will be
316       * moved one hour later into the offset typically corresponding to "summer".
317       * <p>
318       * This method exists primarily for writing test cases.
319       * Non test-code will typically use other methods to create an offset time.
320       * {@code LocalDateTime} has five additional convenience variants of the
321       * equivalent factory method taking fewer arguments.
322       * They are not provided here to reduce the footprint of the API.
323       *
324       * @param year the year to represent, from MIN_YEAR to MAX_YEAR
325       * @param month the month-of-year to represent, from 1 (January) to 12 (December)
326       * @param dayOfMonth the day-of-month to represent, from 1 to 31
327       * @param hour the hour-of-day to represent, from 0 to 23
328       * @param minute the minute-of-hour to represent, from 0 to 59
329       * @param second the second-of-minute to represent, from 0 to 59
330       * @param nanoOfSecond the nano-of-second to represent, from 0 to 999,999,999
331       * @param zone the time-zone, not null

```

```

332     * @return the offset date-time, not null
333     * @throws DateTimeException if the value of any field is out of range, or
334     * if the day-of-month is invalid for the month-year
335     */
336     public static ZonedDateTime of(
337         int year, int month, int dayOfMonth,
338         int hour, int minute, int second, int nanoOfSecond, ZoneId zone) {
339         LocalDateTime dt = LocalDateTime.of(year, month, dayOfMonth, hour, minute, second,
340             nanoOfSecond);
341         return ofLocal(dt, zone, null);
342     }
343
344     /**
345     * Obtains an instance of {@code ZonedDateTime} from a local date-time
346     * using the preferred offset if possible.
347     * <p>
348     * The local date-time is resolved to a single instant on the time-line.
349     * This is achieved by finding a valid offset from UTC/Greenwich for the local
350     * date-time as defined by the {@link ZoneRules rules} of the zone ID.
351     * <p>
352     * In most cases, there is only one valid offset for a local date-time.
353     * In the case of an overlap, where clocks are set back, there are two valid offsets.
354     * If the preferred offset is one of the valid offsets then it is used.
355     * Otherwise the earlier valid offset is used, typically corresponding to "summer".
356     * <p>
357     * In the case of a gap, where clocks jump forward, there is no valid offset.
358     * Instead, the local date-time is adjusted to be later by the length of the gap.
359     * For a typical one hour daylight savings change, the local date-time will be
360     * moved one hour later into the offset typically corresponding to "summer".
361     *
362     * @param localDateTime the local date-time, not null
363     * @param zone the time-zone, not null
364     * @param preferredOffset the zone offset, null if no preference
365     * @return the zoned date-time, not null
366     */
367     public static ZonedDateTime ofLocal(LocalDateTime localDateTime, ZoneId zone, ZoneOffset
368         preferredOffset) {
369         Objects.requireNonNull(localDateTime, "localDateTime");
370         Objects.requireNonNull(zone, "zone");
371         if (zone instanceof ZoneOffset) {
372             return new ZonedDateTime(localDateTime, (ZoneOffset) zone, zone);
373         }
374         ZoneRules rules = zone.getRules();
375         List<ZoneOffset> validOffsets = rules.getValidOffsets(localDateTime);
376         ZoneOffset offset;
377         if (validOffsets.size() == 1) {
378             offset = validOffsets.get(0);
379         } else if (validOffsets.size() == 0) {
380             ZoneOffsetTransition trans = rules.getTransition(localDateTime);
381             localDateTime = localDateTime.plusSeconds(trans.getDuration().getSeconds());
382             offset = trans.getOffsetAfter();
383         } else {
384             if (preferredOffset != null && validOffsets.contains(preferredOffset)) {

```

```

383         offset = preferredOffset;
384     } else {
385         offset = Objects.requireNonNull(validOffsets.get(0), "offset"); // protect against
            bad ZoneRules
386     }
387 }
388 return new ZonedDateTime(localDateTime, offset, zone);
389 }
390
391 //-----
392 /**
393  * Obtains an instance of {@code ZonedDateTime} from an {@code Instant}.
394  * <p>
395  * This creates a zoned date-time with the same instant as that specified.
396  * Calling {@link #toInstant()} will return an instant equal to the one used here.
397  * <p>
398  * Converting an instant to a zoned date-time is simple as there is only one valid
399  * offset for each instant.
400  *
401  * @param instant the instant to create the date-time from, not null
402  * @param zone the time-zone, not null
403  * @return the zoned date-time, not null
404  * @throws DateTimeException if the result exceeds the supported range
405  */
406 public static ZonedDateTime ofInstant(Instant instant, ZoneId zone) {
407     Objects.requireNonNull(instant, "instant");
408     Objects.requireNonNull(zone, "zone");
409     return create(instant.getEpochSecond(), instant.getNano(), zone);
410 }
411
412 /**
413  * Obtains an instance of {@code ZonedDateTime} from the instant formed by combining
414  * the local date-time and offset.
415  * <p>
416  * This creates a zoned date-time by {@link LocalDateTime#toInstant(ZoneOffset) combining}
417  * the {@code LocalDateTime} and {@code ZoneOffset}.
418  * This combination uniquely specifies an instant without ambiguity.
419  * <p>
420  * Converting an instant to a zoned date-time is simple as there is only one valid
421  * offset for each instant. If the valid offset is different to the offset specified,
422  * then the date-time and offset of the zoned date-time will differ from those specified.
423  * <p>
424  * If the {@code ZoneId} to be used is a {@code ZoneOffset}, this method is equivalent
425  * to {@link #of(LocalDateTime, ZoneId)}.
426  *
427  * @param localDateTime the local date-time, not null
428  * @param offset the zone offset, not null
429  * @param zone the time-zone, not null
430  * @return the zoned date-time, not null
431  */
432 public static ZonedDateTime ofInstant(LocalDateTime localDateTime, ZoneOffset offset, ZoneId
    zone) {
433     Objects.requireNonNull(localDateTime, "localDateTime");

```

```

434     Objects.requireNonNull(offset, "offset");
435     Objects.requireNonNull(zone, "zone");
436     if (zone.getRules().isValidOffset(localDateTime, offset)) {
437         return new ZonedDateTime(localDateTime, offset, zone);
438     }
439     return create(localDateTime.toEpochSecond(offset), localDateTime.getNano(), zone);
440 }
441
442 /**
443  * Obtains an instance of {@code ZonedDateTime} using seconds from the
444  * epoch of 1970-01-01T00:00:00Z.
445  *
446  * @param epochSecond the number of seconds from the epoch of 1970-01-01T00:00:00Z
447  * @param nanoOfSecond the nanosecond within the second, from 0 to 999,999,999
448  * @param zone the time-zone, not null
449  * @return the zoned date-time, not null
450  * @throws DateTimeException if the result exceeds the supported range
451  */
452 private static ZonedDateTime create(long epochSecond, int nanoOfSecond, ZoneId zone) {
453     ZoneRules rules = zone.getRules();
454     Instant instant = Instant.ofEpochSecond(epochSecond, nanoOfSecond); // TODO: rules should
455         be queryable by epochSeconds
456     ZoneOffset offset = rules.getOffset(instant);
457     LocalDateTime ldt = LocalDateTime.ofEpochSecond(epochSecond, nanoOfSecond, offset);
458     return new ZonedDateTime(ldt, offset, zone);
459 }
460
461 //-----
462 /**
463  * Obtains an instance of {@code ZonedDateTime} strictly validating the
464  * combination of local date-time, offset and zone ID.
465  *
466  * <p>
467  * This creates a zoned date-time ensuring that the offset is valid for the
468  * local date-time according to the rules of the specified zone.
469  * If the offset is invalid, an exception is thrown.
470  *
471  * @param localDateTime the local date-time, not null
472  * @param offset the zone offset, not null
473  * @param zone the time-zone, not null
474  * @return the zoned date-time, not null
475  */
476 public static ZonedDateTime ofStrict(LocalDateTime localDateTime, ZoneOffset offset, ZoneId
477     zone) {
478     Objects.requireNonNull(localDateTime, "localDateTime");
479     Objects.requireNonNull(offset, "offset");
480     Objects.requireNonNull(zone, "zone");
481     ZoneRules rules = zone.getRules();
482     if (rules.isValidOffset(localDateTime, offset) == false) {
483         ZoneOffsetTransition trans = rules.getTransition(localDateTime);
484         if (trans != null && trans.isGap()) {
485             // error message says daylight savings for simplicity
486             // even though there are other kinds of gaps
487             throw new DateTimeException("LocalDateTime '" + localDateTime +

```

```

485         "' does not exist in zone '" + zone +
486         "' due to a gap in the local time-line, typically caused by daylight
           savings");
487     }
488     throw new DateTimeException("ZoneOffset '" + offset + "' is not valid for LocalDateTime
           '" +
489         localDateTime + "' in zone '" + zone + "'");
490 }
491 return new ZonedDateTime(localDateTime, offset, zone);
492 }
493
494 /**
495  * Obtains an instance of {@code ZonedDateTime} leniently, for advanced use cases,
496  * allowing any combination of local date-time, offset and zone ID.
497  * <p>
498  * This creates a zoned date-time with no checks other than no nulls.
499  * This means that the resulting zoned date-time may have an offset that is in conflict
500  * with the zone ID.
501  * <p>
502  * This method is intended for advanced use cases.
503  * For example, consider the case where a zoned date-time with valid fields is created
504  * and then stored in a database or serialization-based store. At some later point,
505  * the object is then re-loaded. However, between those points in time, the government
506  * that defined the time-zone has changed the rules, such that the originally stored
507  * local date-time now does not occur. This method can be used to create the object
508  * in an "invalid" state, despite the change in rules.
509  *
510  * @param localDateTime the local date-time, not null
511  * @param offset the zone offset, not null
512  * @param zone the time-zone, not null
513  * @return the zoned date-time, not null
514  */
515 private static ZonedDateTime ofLenient(LocalDateTime localDateTime, ZoneOffset offset, ZoneId
    zone) {
516     Objects.requireNonNull(localDateTime, "localDateTime");
517     Objects.requireNonNull(offset, "offset");
518     Objects.requireNonNull(zone, "zone");
519     if (zone instanceof ZoneOffset && offset.equals(zone) == false) {
520         throw new IllegalArgumentException("ZoneId must match ZoneOffset");
521     }
522     return new ZonedDateTime(localDateTime, offset, zone);
523 }
524
525 //-----
526 /**
527  * Obtains an instance of {@code ZonedDateTime} from a temporal object.
528  * <p>
529  * This obtains a zoned date-time based on the specified temporal.
530  * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
531  * which this factory converts to an instance of {@code ZonedDateTime}.
532  * <p>
533  * The conversion will first obtain a {@code ZoneId} from the temporal object,
534  * falling back to a {@code ZoneOffset} if necessary. It will then try to obtain

```

```

535     * an {@code Instant}, falling back to a {@code LocalDateTime} if necessary.
536     * The result will be either the combination of {@code ZoneId} or {@code ZoneOffset}
537     * with {@code Instant} or {@code LocalDateTime}.
538     * Implementations are permitted to perform optimizations such as accessing
539     * those fields that are equivalent to the relevant objects.
540     * <p>
541     * This method matches the signature of the functional interface {@link TemporalQuery}
542     * allowing it to be used as a query via method reference, {@code ZonedDateTime::from}.
543     *
544     * @param temporal the temporal object to convert, not null
545     * @return the zoned date-time, not null
546     * @throws DateTimeException if unable to convert to an {@code ZonedDateTime}
547     */
548     public static ZonedDateTime from(TemporalAccessor temporal) {
549         if (temporal instanceof ZonedDateTime) {
550             return (ZonedDateTime) temporal;
551         }
552         try {
553             ZoneId zone = ZoneId.from(temporal);
554             if (temporal.isSupported(INSTANT_SECONDS)) {
555                 long epochSecond = temporal.getLong(INSTANT_SECONDS);
556                 int nanoOfSecond = temporal.get(NANO_OF_SECOND);
557                 return create(epochSecond, nanoOfSecond, zone);
558             } else {
559                 LocalDate date = LocalDate.from(temporal);
560                 LocalTime time = LocalTime.from(temporal);
561                 return of(date, time, zone);
562             }
563         } catch (DateTimeException ex) {
564             throw new DateTimeException("Unable to obtain ZonedDateTime from TemporalAccessor: " +
565                 temporal + " of type " + temporal.getClass().getName(), ex);
566         }
567     }
568
569     //-----
570     /**
571     * Obtains an instance of {@code ZonedDateTime} from a text string such as
572     * {@code 2007-12-03T10:15:30+01:00[Europe/Paris]}.
573     * <p>
574     * The string must represent a valid date-time and is parsed using
575     * {@link java.time.format.DateTimeFormatter#ISO_ZONED_DATE_TIME}.
576     *
577     * @param text the text to parse such as "2007-12-03T10:15:30+01:00[Europe/Paris]", not null
578     * @return the parsed zoned date-time, not null
579     * @throws DateTimeParseException if the text cannot be parsed
580     */
581     public static ZonedDateTime parse(CharSequence text) {
582         return parse(text, DateTimeFormatter.ISO_ZONED_DATE_TIME);
583     }
584
585     /**
586     * Obtains an instance of {@code ZonedDateTime} from a text string using a specific formatter.
587     * <p>

```

```

588     * The text is parsed using the formatter, returning a date-time.
589     *
590     * @param text    the text to parse, not null
591     * @param formatter the formatter to use, not null
592     * @return the parsed zoned date-time, not null
593     * @throws DateTimeParseException if the text cannot be parsed
594     */
595     public static ZonedDateTime parse(CharSequence text, DateTimeFormatter formatter) {
596         Objects.requireNonNull(formatter, "formatter");
597         return formatter.parse(text, ZonedDateTime::from);
598     }
599
600     //-----
601     /**
602     * Constructor.
603     *
604     * @param dateTime the date-time, validated as not null
605     * @param offset    the zone offset, validated as not null
606     * @param zone      the time-zone, validated as not null
607     */
608     private ZonedDateTime(LocalDateTime dateTime, ZoneOffset offset, ZoneId zone) {
609         this.dateTime = dateTime;
610         this.offset = offset;
611         this.zone = zone;
612     }
613
614     /**
615     * Resolves the new local date-time using this zone ID, retaining the offset if possible.
616     *
617     * @param newDateTime the new local date-time, not null
618     * @return the zoned date-time, not null
619     */
620     private ZonedDateTime resolveLocal(LocalDateTime newDateTime) {
621         return ofLocal(newDateTime, zone, offset);
622     }
623
624     /**
625     * Resolves the new local date-time using the offset to identify the instant.
626     *
627     * @param newDateTime the new local date-time, not null
628     * @return the zoned date-time, not null
629     */
630     private ZonedDateTime resolveInstant(LocalDateTime newDateTime) {
631         return ofInstant(newDateTime, offset, zone);
632     }
633
634     /**
635     * Resolves the offset into this zoned date-time for the with methods.
636     * <p>
637     * This typically ignores the offset, unless it can be used to switch offset in a DST overlap.
638     *
639     * @param offset the offset, not null
640     * @return the zoned date-time, not null

```



```

641     */
642     private ZonedDateTime resolveOffset(ZoneOffset offset) {
643         if (offset.equals(this.offset) == false && zone.getRules().isValidOffset(dateTime, offset))
644             {
645                 return new ZonedDateTime(dateTime, offset, zone);
646             }
647         return this;
648     }
649
650     //-----
651     /**
652      * Checks if the specified field is supported.
653      * <p>
654      * This checks if this date-time can be queried for the specified field.
655      * If false, then calling the {@link #range(TemporalField) range},
656      * {@link #get(TemporalField) get} and {@link #with(TemporalField, long)}
657      * methods will throw an exception.
658      * <p>
659      * If the field is a {@link ChronoField} then the query is implemented here.
660      * The supported fields are:
661      * <ul>
662      * <li>{@code NANO_OF_SECOND}
663      * <li>{@code NANO_OF_DAY}
664      * <li>{@code MICRO_OF_SECOND}
665      * <li>{@code MICRO_OF_DAY}
666      * <li>{@code MILLI_OF_SECOND}
667      * <li>{@code MILLI_OF_DAY}
668      * <li>{@code SECOND_OF_MINUTE}
669      * <li>{@code SECOND_OF_DAY}
670      * <li>{@code MINUTE_OF_HOUR}
671      * <li>{@code MINUTE_OF_DAY}
672      * <li>{@code HOUR_OF_AMPM}
673      * <li>{@code CLOCK_HOUR_OF_AMPM}
674      * <li>{@code HOUR_OF_DAY}
675      * <li>{@code CLOCK_HOUR_OF_DAY}
676      * <li>{@code AMPM_OF_DAY}
677      * <li>{@code DAY_OF_WEEK}
678      * <li>{@code ALIGNED_DAY_OF_WEEK_IN_MONTH}
679      * <li>{@code ALIGNED_DAY_OF_WEEK_IN_YEAR}
680      * <li>{@code DAY_OF_MONTH}
681      * <li>{@code DAY_OF_YEAR}
682      * <li>{@code EPOCH_DAY}
683      * <li>{@code ALIGNED_WEEK_OF_MONTH}
684      * <li>{@code ALIGNED_WEEK_OF_YEAR}
685      * <li>{@code MONTH_OF_YEAR}
686      * <li>{@code PROLEPTIC_MONTH}
687      * <li>{@code YEAR_OF_ERA}
688      * <li>{@code YEAR}
689      * <li>{@code ERA}
690      * <li>{@code INSTANT_SECONDS}
691      * <li>{@code OFFSET_SECONDS}
692      * </ul>
693      * All other {@code ChronoField} instances will return false.

```



```

693     * <p>
694     * If the field is not a {@code ChronoField}, then the result of this method
695     * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
696     * passing {@code this} as the argument.
697     * Whether the field is supported is determined by the field.
698     *
699     * @param field the field to check, null returns false
700     * @return true if the field is supported on this date-time, false if not
701     */
702     @Override
703     public boolean isSupported(TemporalField field) {
704         return field instanceof ChronoField || (field != null && field.isSupportedBy(this));
705     }
706
707     /**
708     * Checks if the specified unit is supported.
709     * <p>
710     * This checks if the specified unit can be added to, or subtracted from, this date-time.
711     * If false, then calling the {@link #plus(long, TemporalUnit)} and
712     * {@link #minus(long, TemporalUnit) minus} methods will throw an exception.
713     * <p>
714     * If the unit is a {@link ChronoUnit} then the query is implemented here.
715     * The supported units are:
716     * <ul>
717     * <li>{@code NANOS}
718     * <li>{@code MICROS}
719     * <li>{@code MILLIS}
720     * <li>{@code SECONDS}
721     * <li>{@code MINUTES}
722     * <li>{@code HOURS}
723     * <li>{@code HALF_DAYS}
724     * <li>{@code DAYS}
725     * <li>{@code WEEKS}
726     * <li>{@code MONTHS}
727     * <li>{@code YEARS}
728     * <li>{@code DECADES}
729     * <li>{@code CENTURIES}
730     * <li>{@code MILLENNIA}
731     * <li>{@code ERAS}
732     * </ul>
733     * All other {@code ChronoUnit} instances will return false.
734     * <p>
735     * If the unit is not a {@code ChronoUnit}, then the result of this method
736     * is obtained by invoking {@code TemporalUnit.isSupportedBy(Temporal)}
737     * passing {@code this} as the argument.
738     * Whether the unit is supported is determined by the unit.
739     *
740     * @param unit the unit to check, null returns false
741     * @return true if the unit can be added/subtracted, false if not
742     */
743     @Override // override for Javadoc
744     public boolean isSupported(TemporalUnit unit) {
745         return ChronoZonedDateTime.super.isSupported(unit);

```

```

746     }
747
748     //-----
749     /**
750      * Gets the range of valid values for the specified field.
751      * <p>
752      * The range object expresses the minimum and maximum valid values for a field.
753      * This date-time is used to enhance the accuracy of the returned range.
754      * If it is not possible to return the range, because the field is not supported
755      * or for some other reason, an exception is thrown.
756      * <p>
757      * If the field is a {@link ChronoField} then the query is implemented here.
758      * The {@link #isSupported(TemporalField) supported fields} will return
759      * appropriate range instances.
760      * All other {@code ChronoField} instances will throw an {@code
761      *     UnsupportedTemporalTypeException}.
762      * <p>
763      * If the field is not a {@code ChronoField}, then the result of this method
764      * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
765      * passing {@code this} as the argument.
766      * Whether the range can be obtained is determined by the field.
767      *
768      * @param field the field to query the range for, not null
769      * @return the range of valid values for the field, not null
770      * @throws DateTimeException if the range for the field cannot be obtained
771      * @throws UnsupportedTemporalTypeException if the field is not supported
772      */
773     @Override
774     public ValueRange range(TemporalField field) {
775         if (field instanceof ChronoField) {
776             if (field == INSTANT_SECONDS || field == OFFSET_SECONDS) {
777                 return field.range();
778             }
779             return dateTime.range(field);
780         }
781         return field.rangeRefinedBy(this);
782     }
783
784     /**
785      * Gets the value of the specified field from this date-time as an {@code int}.
786      * <p>
787      * This queries this date-time for the value of the specified field.
788      * The returned value will always be within the valid range of values for the field.
789      * If it is not possible to return the value, because the field is not supported
790      * or for some other reason, an exception is thrown.
791      * <p>
792      * If the field is a {@link ChronoField} then the query is implemented here.
793      * The {@link #isSupported(TemporalField) supported fields} will return valid
794      * values based on this date-time, except {@code NANO_OF_DAY}, {@code MICRO_OF_DAY},
795      * {@code EPOCH_DAY}, {@code PROLEPTIC_MONTH} and {@code INSTANT_SECONDS} which are too
796      * large to fit in an {@code int} and throw a {@code DateTimeException}.
797      * All other {@code ChronoField} instances will throw an {@code
798      *     UnsupportedTemporalTypeException}.

```

```

797 * <p>
798 * If the field is not a {@code ChronoField}, then the result of this method
799 * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
800 * passing {@code this} as the argument. Whether the value can be obtained,
801 * and what the value represents, is determined by the field.
802 *
803 * @param field the field to get, not null
804 * @return the value for the field
805 * @throws DateTimeException if a value for the field cannot be obtained or
806 *         the value is outside the range of valid values for the field
807 * @throws UnsupportedTemporalTypeException if the field is not supported or
808 *         the range of values exceeds an {@code int}
809 * @throws ArithmeticException if numeric overflow occurs
810 */
811 @Override // override for Javadoc and performance
812 public int get(TemporalField field) {
813     if (field instanceof ChronoField) {
814         switch ((ChronoField) field) {
815             case INSTANT_SECONDS:
816                 throw new UnsupportedTemporalTypeException("Invalid field 'InstantSeconds' for
                        get() method, use getLong() instead");
817             case OFFSET_SECONDS:
818                 return getOffset().getTotalSeconds();
819         }
820     }
821     return dateTime.get(field);
822 }
823 }
824
825 /**
826 * Gets the value of the specified field from this date-time as a {@code long}.
827 * <p>
828 * This queries this date-time for the value of the specified field.
829 * If it is not possible to return the value, because the field is not supported
830 * or for some other reason, an exception is thrown.
831 * <p>
832 * If the field is a {@link ChronoField} then the query is implemented here.
833 * The {@link #isSupported(TemporalField) supported fields} will return valid
834 * values based on this date-time.
835 * All other {@code ChronoField} instances will throw an {@code
        UnsupportedTemporalTypeException}.
836 * <p>
837 * If the field is not a {@code ChronoField}, then the result of this method
838 * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
839 * passing {@code this} as the argument. Whether the value can be obtained,
840 * and what the value represents, is determined by the field.
841 *
842 * @param field the field to get, not null
843 * @return the value for the field
844 * @throws DateTimeException if a value for the field cannot be obtained
845 * @throws UnsupportedTemporalTypeException if the field is not supported
846 * @throws ArithmeticException if numeric overflow occurs
847 */

```

```

848     @Override
849     public long getLong(TemporalField field) {
850         if (field instanceof ChronoField) {
851             switch ((ChronoField) field) {
852                 case INSTANT_SECONDS: return toEpochSecond();
853                 case OFFSET_SECONDS: return getOffset().getTotalSeconds();
854             }
855             return dateTime.getLong(field);
856         }
857         return field.getFrom(this);
858     }
859
860     //-----
861     /**
862      * Gets the zone offset, such as '+01:00'.
863      * <p>
864      * This is the offset of the local date-time from UTC/Greenwich.
865      *
866      * @return the zone offset, not null
867      */
868     @Override
869     public ZoneOffset getOffset() {
870         return offset;
871     }
872
873     /**
874      * Returns a copy of this date-time changing the zone offset to the
875      * earlier of the two valid offsets at a local time-line overlap.
876      * <p>
877      * This method only has any effect when the local time-line overlaps, such as
878      * at an autumn daylight savings cutover. In this scenario, there are two
879      * valid offsets for the local date-time. Calling this method will return
880      * a zoned date-time with the earlier of the two selected.
881      * <p>
882      * If this method is called when it is not an overlap, {@code this}
883      * is returned.
884      * <p>
885      * This instance is immutable and unaffected by this method call.
886      *
887      * @return a {@code ZonedDateTime} based on this date-time with the earlier offset, not null
888      */
889     @Override
890     public ZonedDateTime withEarlierOffsetAtOverlap() {
891         ZoneOffsetTransition trans = getZone().getRules().getTransition(dateTime);
892         if (trans != null && trans.isOverlap()) {
893             ZoneOffset earlierOffset = trans.getOffsetBefore();
894             if (earlierOffset.equals(offset) == false) {
895                 return new ZonedDateTime(dateTime, earlierOffset, zone);
896             }
897         }
898         return this;
899     }
900

```

```

901 /**
902  * Returns a copy of this date-time changing the zone offset to the
903  * later of the two valid offsets at a local time-line overlap.
904  * <p>
905  * This method only has any effect when the local time-line overlaps, such as
906  * at an autumn daylight savings cutover. In this scenario, there are two
907  * valid offsets for the local date-time. Calling this method will return
908  * a zoned date-time with the later of the two selected.
909  * <p>
910  * If this method is called when it is not an overlap, {@code this}
911  * is returned.
912  * <p>
913  * This instance is immutable and unaffected by this method call.
914  *
915  * @return a {@code ZonedDateTime} based on this date-time with the later offset, not null
916  */
917 @Override
918 public ZonedDateTime withLaterOffsetAtOverlap() {
919     ZoneOffsetTransition trans = getZone().getRules().getTransition(toLocalDateTime());
920     if (trans != null) {
921         ZoneOffset laterOffset = trans.getOffsetAfter();
922         if (laterOffset.equals(offset) == false) {
923             return new ZonedDateTime(dateTime, laterOffset, zone);
924         }
925     }
926     return this;
927 }
928
929 //-----
930 /**
931  * Gets the time-zone, such as 'Europe/Paris'.
932  * <p>
933  * This returns the zone ID. This identifies the time-zone {@link ZoneRules rules}
934  * that determine when and how the offset from UTC/Greenwich changes.
935  * <p>
936  * The zone ID may be same as the {@linkplain #getOffset() offset}.
937  * If this is true, then any future calculations, such as addition or subtraction,
938  * have no complex edge cases due to time-zone rules.
939  * See also {@link #withFixedOffsetZone()}.
940  *
941  * @return the time-zone, not null
942  */
943 @Override
944 public ZoneId getZone() {
945     return zone;
946 }
947
948 /**
949  * Returns a copy of this date-time with a different time-zone,
950  * retaining the local date-time if possible.
951  * <p>
952  * This method changes the time-zone and retains the local date-time.
953  * The local date-time is only changed if it is invalid for the new zone,

```

```

954     * determined using the same approach as
955     * {@link #ofLocal(LocalDateTime, ZoneId, ZoneOffset)}.
956     * <p>
957     * To change the zone and adjust the local date-time,
958     * use {@link #withZoneSameInstant(ZoneId)}.
959     * <p>
960     * This instance is immutable and unaffected by this method call.
961     *
962     * @param zone the time-zone to change to, not null
963     * @return a {@code ZonedDateTime} based on this date-time with the requested zone, not null
964     */
965     @Override
966     public ZonedDateTime withZoneSameLocal(ZoneId zone) {
967         Objects.requireNonNull(zone, "zone");
968         return this.zone.equals(zone) ? this : ofLocal(dateTime, zone, offset);
969     }
970
971     /**
972     * Returns a copy of this date-time with a different time-zone,
973     * retaining the instant.
974     * <p>
975     * This method changes the time-zone and retains the instant.
976     * This normally results in a change to the local date-time.
977     * <p>
978     * This method is based on retaining the same instant, thus gaps and overlaps
979     * in the local time-line have no effect on the result.
980     * <p>
981     * To change the offset while keeping the local time,
982     * use {@link #withZoneSameLocal(ZoneId)}.
983     *
984     * @param zone the time-zone to change to, not null
985     * @return a {@code ZonedDateTime} based on this date-time with the requested zone, not null
986     * @throws DateTimeException if the result exceeds the supported date range
987     */
988     @Override
989     public ZonedDateTime withZoneSameInstant(ZoneId zone) {
990         Objects.requireNonNull(zone, "zone");
991         return this.zone.equals(zone) ? this :
992             create(dateTime.toEpochSecond(offset), dateTime.getNano(), zone);
993     }
994
995     /**
996     * Returns a copy of this date-time with the zone ID set to the offset.
997     * <p>
998     * This returns a zoned date-time where the zone ID is the same as {@link #getOffset()}.
999     * The local date-time, offset and instant of the result will be the same as in this date-time.
1000     * <p>
1001     * Setting the date-time to a fixed single offset means that any future
1002     * calculations, such as addition or subtraction, have no complex edge cases
1003     * due to time-zone rules.
1004     * This might also be useful when sending a zoned date-time across a network,
1005     * as most protocols, such as ISO-8601, only handle offsets,
1006     * and not region-based zone IDs.

```

```

1007 * <p>
1008 * This is equivalent to {@code ZonedDateTime.of(zdt.toLocalDateTime(), zdt.getOffset())}.
1009 *
1010 * @return a {@code ZonedDateTime} with the zone ID set to the offset, not null
1011 */
1012 public ZonedDateTime withFixedOffsetZone() {
1013     return this.zone.equals(offset) ? this : new ZonedDateTime(dateTime, offset, offset);
1014 }
1015
1016 //-----
1017 /**
1018 * Gets the {@code LocalDateTime} part of this date-time.
1019 * <p>
1020 * This returns a {@code LocalDateTime} with the same year, month, day and time
1021 * as this date-time.
1022 *
1023 * @return the local date-time part of this date-time, not null
1024 */
1025 @Override // override for return type
1026 public LocalDateTime toLocalDateTime() {
1027     return dateTime;
1028 }
1029
1030 //-----
1031 /**
1032 * Gets the {@code LocalDate} part of this date-time.
1033 * <p>
1034 * This returns a {@code LocalDate} with the same year, month and day
1035 * as this date-time.
1036 *
1037 * @return the date part of this date-time, not null
1038 */
1039 @Override // override for return type
1040 public LocalDate toLocalDate() {
1041     return dateTime.toLocalDate();
1042 }
1043
1044 /**
1045 * Gets the year field.
1046 * <p>
1047 * This method returns the primitive {@code int} value for the year.
1048 * <p>
1049 * The year returned by this method is proleptic as per {@code get(YEAR)}.
1050 * To obtain the year-of-era, use {@code get(YEAR_OF_ERA)}.
1051 *
1052 * @return the year, from MIN_YEAR to MAX_YEAR
1053 */
1054 public int getYear() {
1055     return dateTime.getYear();
1056 }
1057
1058 /**
1059 * Gets the month-of-year field from 1 to 12.

```

```
1060 * <p>
1061 * This method returns the month as an {@code int} from 1 to 12.
1062 * Application code is frequently clearer if the enum {@link Month}
1063 * is used by calling {@link #getMonth()}.
1064 *
1065 * @return the month-of-year, from 1 to 12
1066 * @see #getMonth()
1067 */
1068 public int getMonthValue() {
1069     return dateTime.getMonthValue();
1070 }
1071
1072 /**
1073 * Gets the month-of-year field using the {@code Month} enum.
1074 * <p>
1075 * This method returns the enum {@link Month} for the month.
1076 * This avoids confusion as to what {@code int} values mean.
1077 * If you need access to the primitive {@code int} value then the enum
1078 * provides the {@link Month#getValue() int value}.
1079 *
1080 * @return the month-of-year, not null
1081 * @see #getMonthValue()
1082 */
1083 public Month getMonth() {
1084     return dateTime.getMonth();
1085 }
1086
1087 /**
1088 * Gets the day-of-month field.
1089 * <p>
1090 * This method returns the primitive {@code int} value for the day-of-month.
1091 *
1092 * @return the day-of-month, from 1 to 31
1093 */
1094 public int getDayOfMonth() {
1095     return dateTime.getDayOfMonth();
1096 }
1097
1098 /**
1099 * Gets the day-of-year field.
1100 * <p>
1101 * This method returns the primitive {@code int} value for the day-of-year.
1102 *
1103 * @return the day-of-year, from 1 to 365, or 366 in a leap year
1104 */
1105 public int getDayOfYear() {
1106     return dateTime.getDayOfYear();
1107 }
1108
1109 /**
1110 * Gets the day-of-week field, which is an enum {@code DayOfWeek}.
1111 * <p>
1112 * This method returns the enum {@link DayOfWeek} for the day-of-week.
```



```
1113     * This avoids confusion as to what {@code int} values mean.
1114     * If you need access to the primitive {@code int} value then the enum
1115     * provides the {@link DayOfWeek#getValue() int value}.
1116     * <p>
1117     * Additional information can be obtained from the {@code DayOfWeek}.
1118     * This includes textual names of the values.
1119     *
1120     * @return the day-of-week, not null
1121     */
1122     public DayOfWeek getDayOfWeek() {
1123         return dateTime.getDayOfWeek();
1124     }
1125
1126     //-----
1127     /**
1128     * Gets the {@code LocalTime} part of this date-time.
1129     * <p>
1130     * This returns a {@code LocalTime} with the same hour, minute, second and
1131     * nanosecond as this date-time.
1132     *
1133     * @return the time part of this date-time, not null
1134     */
1135     @Override // override for Javadoc and performance
1136     public LocalTime toLocalTime() {
1137         return dateTime.toLocalTime();
1138     }
1139
1140     /**
1141     * Gets the hour-of-day field.
1142     *
1143     * @return the hour-of-day, from 0 to 23
1144     */
1145     public int getHour() {
1146         return dateTime.getHour();
1147     }
1148
1149     /**
1150     * Gets the minute-of-hour field.
1151     *
1152     * @return the minute-of-hour, from 0 to 59
1153     */
1154     public int getMinute() {
1155         return dateTime.getMinute();
1156     }
1157
1158     /**
1159     * Gets the second-of-minute field.
1160     *
1161     * @return the second-of-minute, from 0 to 59
1162     */
1163     public int getSecond() {
1164         return dateTime.getSecond();
1165     }
```

```

1166
1167 /**
1168  * Gets the nano-of-second field.
1169  *
1170  * @return the nano-of-second, from 0 to 999,999,999
1171  */
1172 public int getNano() {
1173     return dateTime.getNano();
1174 }
1175
1176 //-----
1177 /**
1178  * Returns an adjusted copy of this date-time.
1179  * <p>
1180  * This returns a {@code ZonedDateTime}, based on this one, with the date-time adjusted.
1181  * The adjustment takes place using the specified adjuster strategy object.
1182  * Read the documentation of the adjuster to understand what adjustment will be made.
1183  * <p>
1184  * A simple adjuster might simply set the one of the fields, such as the year field.
1185  * A more complex adjuster might set the date to the last day of the month.
1186  * A selection of common adjustments is provided in
1187  * {@link java.time.temporal.TemporalAdjusters TemporalAdjusters}.
1188  * These include finding the "last day of the month" and "next Wednesday".
1189  * Key date-time classes also implement the {@code TemporalAdjuster} interface,
1190  * such as {@link Month} and {@link java.time.MonthDay MonthDay}.
1191  * The adjuster is responsible for handling special cases, such as the varying
1192  * lengths of month and leap years.
1193  * <p>
1194  * For example this code returns a date on the last day of July:
1195  * <pre>
1196  * import static java.time.Month.*;
1197  * import static java.time.temporal.TemporalAdjusters.*;
1198  *
1199  * result = zonedDateTime.with(JULY).with(lastDayOfMonth());
1200  * </pre>
1201  * <p>
1202  * The classes {@link LocalDate} and {@link LocalTime} implement {@code TemporalAdjuster},
1203  * thus this method can be used to change the date, time or offset:
1204  * <pre>
1205  * result = zonedDateTime.with(date);
1206  * result = zonedDateTime.with(time);
1207  * </pre>
1208  * <p>
1209  * {@link ZoneOffset} also implements {@code TemporalAdjuster} however using it
1210  * as an argument typically has no effect. The offset of a {@code ZonedDateTime} is
1211  * controlled primarily by the time-zone. As such, changing the offset does not generally
1212  * make sense, because there is only one valid offset for the local date-time and zone.
1213  * If the zoned date-time is in a daylight savings overlap, then the offset is used
1214  * to switch between the two valid offsets. In all other cases, the offset is ignored.
1215  * <p>
1216  * The result of this method is obtained by invoking the
1217  * {@link TemporalAdjuster#adjustInto(Temporal)} method on the
1218  * specified adjuster passing {@code this} as the argument.

```

```

1219 * <p>
1220 * This instance is immutable and unaffected by this method call.
1221 *
1222 * @param adjuster the adjuster to use, not null
1223 * @return a {@code ZonedDateTime} based on {@code this} with the adjustment made, not null
1224 * @throws DateTimeException if the adjustment cannot be made
1225 * @throws ArithmeticException if numeric overflow occurs
1226 */
1227 @Override
1228 public ZonedDateTime with(TemporalAdjuster adjuster) {
1229     // optimizations
1230     if (adjuster instanceof LocalDate) {
1231         return resolveLocal(LocalDateTime.of((LocalDate) adjuster, dateTime.toLocalTime()));
1232     } else if (adjuster instanceof LocalTime) {
1233         return resolveLocal(LocalDateTime.of(dateTime.toLocalDate(), (LocalTime) adjuster));
1234     } else if (adjuster instanceof LocalDateTime) {
1235         return resolveLocal((LocalDateTime) adjuster);
1236     } else if (adjuster instanceof OffsetDateTime) {
1237         OffsetDateTime odt = (OffsetDateTime) adjuster;
1238         return ofLocal(odt.toLocalDateTime(), zone, odt.getOffset());
1239     } else if (adjuster instanceof Instant) {
1240         Instant instant = (Instant) adjuster;
1241         return create(instant.getEpochSecond(), instant.getNano(), zone);
1242     } else if (adjuster instanceof ZoneOffset) {
1243         return resolveOffset((ZoneOffset) adjuster);
1244     }
1245     return (ZonedDateTime) adjuster.adjustInto(this);
1246 }
1247
1248 /**
1249 * Returns a copy of this date-time with the specified field set to a new value.
1250 * <p>
1251 * This returns a {@code ZonedDateTime}, based on this one, with the value
1252 * for the specified field changed.
1253 * This can be used to change any supported field, such as the year, month or day-of-month.
1254 * If it is not possible to set the value, because the field is not supported or for
1255 * some other reason, an exception is thrown.
1256 * <p>
1257 * In some cases, changing the specified field can cause the resulting date-time to become
1258 * invalid,
1259 * such as changing the month from 31st January to February would make the day-of-month invalid
1260 * .
1261 * In cases like this, the field is responsible for resolving the date. Typically it will
1262 * choose
1263 * the previous valid date, which would be the last valid day of February in this example.
1264 * <p>
1265 * If the field is a {@link ChronoField} then the adjustment is implemented here.
1266 * <p>
1267 * The {@code INSTANT_SECONDS} field will return a date-time with the specified instant.
1268 * The zone and nano-of-second are unchanged.
1269 * The result will have an offset derived from the new instant and original zone.
1270 * If the new instant value is outside the valid range then a {@code DateTimeException} will be
1271 * thrown.

```

```

1268 * <p>
1269 * The {@code OFFSET_SECONDS} field will typically be ignored.
1270 * The offset of a {@code ZonedDateTime} is controlled primarily by the time-zone.
1271 * As such, changing the offset does not generally make sense, because there is only
1272 * one valid offset for the local date-time and zone.
1273 * If the zoned date-time is in a daylight savings overlap, then the offset is used
1274 * to switch between the two valid offsets. In all other cases, the offset is ignored.
1275 * If the new offset value is outside the valid range then a {@code DateTimeException} will be
    thrown.
1276 * <p>
1277 * The other {@link #isSupported(TemporalField) supported fields} will behave as per
1278 * the matching method on {@link LocalDateTime#with(TemporalField, long) LocalDateTime}.
1279 * The zone is not part of the calculation and will be unchanged.
1280 * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1281 * then the offset will be retained if possible, otherwise the earlier offset will be used.
1282 * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1283 * <p>
1284 * All other {@code ChronoField} instances will throw an {@code
    UnsupportedTemporalTypeException}.
1285 * <p>
1286 * If the field is not a {@code ChronoField}, then the result of this method
1287 * is obtained by invoking {@code TemporalField.adjustInto(Temporal, long)}
1288 * passing {@code this} as the argument. In this case, the field determines
1289 * whether and how to adjust the instant.
1290 * <p>
1291 * This instance is immutable and unaffected by this method call.
1292 *
1293 * @param field the field to set in the result, not null
1294 * @param newValue the new value of the field in the result
1295 * @return a {@code ZonedDateTime} based on {@code this} with the specified field set, not null
1296 * @throws DateTimeException if the field cannot be set
1297 * @throws UnsupportedTemporalTypeException if the field is not supported
1298 * @throws ArithmeticException if numeric overflow occurs
1299 */
1300 @Override
1301 public ZonedDateTime with(TemporalField field, long newValue) {
1302     if (field instanceof ChronoField) {
1303         ChronoField f = (ChronoField) field;
1304         switch (f) {
1305             case INSTANT_SECONDS:
1306                 return create(newValue, getNano(), zone);
1307             case OFFSET_SECONDS:
1308                 ZoneOffset offset = ZoneOffset.ofTotalSeconds(f.checkValidIntValue(newValue));
1309                 return resolveOffset(offset);
1310         }
1311         return resolveLocal(dateTime.with(field, newValue));
1312     }
1313     return field.adjustInto(this, newValue);
1314 }
1315
1316 //-----
1317 /**
1318 * Returns a copy of this {@code ZonedDateTime} with the year altered.

```

```
1319 * <p>
1320 * This operates on the local time-line,
1321 * {@link LocalDateTime#withYear(int) changing the year} of the local date-time.
1322 * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1323 * to obtain the offset.
1324 * <p>
1325 * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1326 * then the offset will be retained if possible, otherwise the earlier offset will be used.
1327 * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1328 * <p>
1329 * This instance is immutable and unaffected by this method call.
1330 *
1331 * @param year the year to set in the result, from MIN_YEAR to MAX_YEAR
1332 * @return a {@code ZonedDateTime} based on this date-time with the requested year, not null
1333 * @throws DateTimeException if the year value is invalid
1334 */
1335 public ZonedDateTime withYear(int year) {
1336     return resolveLocal(dateTime.withYear(year));
1337 }
1338
1339 /**
1340 * Returns a copy of this {@code ZonedDateTime} with the month-of-year altered.
1341 * <p>
1342 * This operates on the local time-line,
1343 * {@link LocalDateTime#withMonth(int) changing the month} of the local date-time.
1344 * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1345 * to obtain the offset.
1346 * <p>
1347 * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1348 * then the offset will be retained if possible, otherwise the earlier offset will be used.
1349 * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1350 * <p>
1351 * This instance is immutable and unaffected by this method call.
1352 *
1353 * @param month the month-of-year to set in the result, from 1 (January) to 12 (December)
1354 * @return a {@code ZonedDateTime} based on this date-time with the requested month, not null
1355 * @throws DateTimeException if the month-of-year value is invalid
1356 */
1357 public ZonedDateTime withMonth(int month) {
1358     return resolveLocal(dateTime.withMonth(month));
1359 }
1360
1361 /**
1362 * Returns a copy of this {@code ZonedDateTime} with the day-of-month altered.
1363 * <p>
1364 * This operates on the local time-line,
1365 * {@link LocalDateTime#withDayOfMonth(int) changing the day-of-month} of the local date-time.
1366 * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1367 * to obtain the offset.
1368 * <p>
1369 * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1370 * then the offset will be retained if possible, otherwise the earlier offset will be used.
1371 * If in a gap, the local date-time will be adjusted forward by the length of the gap.
```

```

1372 * <p>
1373 * This instance is immutable and unaffected by this method call.
1374 *
1375 * @param dayOfMonth the day-of-month to set in the result, from 1 to 28-31
1376 * @return a {@code ZonedDateTime} based on this date-time with the requested day, not null
1377 * @throws DateTimeException if the day-of-month value is invalid,
1378 * or if the day-of-month is invalid for the month-year
1379 */
1380 public ZonedDateTime withDayOfMonth(int dayOfMonth) {
1381     return resolveLocal(dateTime.withDayOfMonth(dayOfMonth));
1382 }
1383
1384 /**
1385 * Returns a copy of this {@code ZonedDateTime} with the day-of-year altered.
1386 * <p>
1387 * This operates on the local time-line,
1388 * {@link LocalDateTime#withDayOfYear(int)} changing the day-of-year of the local date-time.
1389 * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1390 * to obtain the offset.
1391 * <p>
1392 * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1393 * then the offset will be retained if possible, otherwise the earlier offset will be used.
1394 * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1395 * <p>
1396 * This instance is immutable and unaffected by this method call.
1397 *
1398 * @param dayOfYear the day-of-year to set in the result, from 1 to 365-366
1399 * @return a {@code ZonedDateTime} based on this date with the requested day, not null
1400 * @throws DateTimeException if the day-of-year value is invalid,
1401 * or if the day-of-year is invalid for the year
1402 */
1403 public ZonedDateTime withDayOfYear(int dayOfYear) {
1404     return resolveLocal(dateTime.withDayOfYear(dayOfYear));
1405 }
1406
1407 //-----
1408 /**
1409 * Returns a copy of this {@code ZonedDateTime} with the hour-of-day altered.
1410 * <p>
1411 * This operates on the local time-line,
1412 * {@link LocalDateTime#withHour(int)} changing the time of the local date-time.
1413 * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1414 * to obtain the offset.
1415 * <p>
1416 * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1417 * then the offset will be retained if possible, otherwise the earlier offset will be used.
1418 * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1419 * <p>
1420 * This instance is immutable and unaffected by this method call.
1421 *
1422 * @param hour the hour-of-day to set in the result, from 0 to 23
1423 * @return a {@code ZonedDateTime} based on this date-time with the requested hour, not null
1424 * @throws DateTimeException if the hour value is invalid

```

```
1425     */
1426     public ZonedDateTime withHour(int hour) {
1427         return resolveLocal(dateTime.withHour(hour));
1428     }
1429
1430     /**
1431      * Returns a copy of this {@code ZonedDateTime} with the minute-of-hour altered.
1432      * <p>
1433      * This operates on the local time-line,
1434      * {@linkplain LocalDateTime#withMinute(int) changing the time} of the local date-time.
1435      * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1436      * to obtain the offset.
1437      * <p>
1438      * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1439      * then the offset will be retained if possible, otherwise the earlier offset will be used.
1440      * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1441      * <p>
1442      * This instance is immutable and unaffected by this method call.
1443      *
1444      * @param minute the minute-of-hour to set in the result, from 0 to 59
1445      * @return a {@code ZonedDateTime} based on this date-time with the requested minute, not null
1446      * @throws DateTimeException if the minute value is invalid
1447      */
1448     public ZonedDateTime withMinute(int minute) {
1449         return resolveLocal(dateTime.withMinute(minute));
1450     }
1451
1452     /**
1453      * Returns a copy of this {@code ZonedDateTime} with the second-of-minute altered.
1454      * <p>
1455      * This operates on the local time-line,
1456      * {@linkplain LocalDateTime#withSecond(int) changing the time} of the local date-time.
1457      * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1458      * to obtain the offset.
1459      * <p>
1460      * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1461      * then the offset will be retained if possible, otherwise the earlier offset will be used.
1462      * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1463      * <p>
1464      * This instance is immutable and unaffected by this method call.
1465      *
1466      * @param second the second-of-minute to set in the result, from 0 to 59
1467      * @return a {@code ZonedDateTime} based on this date-time with the requested second, not null
1468      * @throws DateTimeException if the second value is invalid
1469      */
1470     public ZonedDateTime withSecond(int second) {
1471         return resolveLocal(dateTime.withSecond(second));
1472     }
1473
1474     /**
1475      * Returns a copy of this {@code ZonedDateTime} with the nano-of-second altered.
1476      * <p>
1477      * This operates on the local time-line,
```



```

1478 * {@linkplain LocalDateTime#withNano(int) changing the time} of the local date-time.
1479 * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1480 * to obtain the offset.
1481 * <p>
1482 * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1483 * then the offset will be retained if possible, otherwise the earlier offset will be used.
1484 * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1485 * <p>
1486 * This instance is immutable and unaffected by this method call.
1487 *
1488 * @param nanoOfSecond the nano-of-second to set in the result, from 0 to 999,999,999
1489 * @return a {@code ZonedDateTime} based on this date-time with the requested nanosecond, not
1490 *         null
1491 * @throws DateTimeException if the nano value is invalid
1492 */
1493 public ZonedDateTime withNano(int nanoOfSecond) {
1494     return resolveLocal(dateTime.withNano(nanoOfSecond));
1495 }
1496
1497 //-----
1498 /**
1499 * Returns a copy of this {@code ZonedDateTime} with the time truncated.
1500 * <p>
1501 * Truncation returns a copy of the original date-time with fields
1502 * smaller than the specified unit set to zero.
1503 * For example, truncating with the {@link ChronoUnit#MINUTES minutes} unit
1504 * will set the second-of-minute and nano-of-second field to zero.
1505 * <p>
1506 * The unit must have a {@linkplain TemporalUnit#getDuration() duration}
1507 * that divides into the length of a standard day without remainder.
1508 * This includes all supplied time units on {@link ChronoUnit} and
1509 * {@link ChronoUnit#DAYS DAYS}. Other units throw an exception.
1510 * <p>
1511 * This operates on the local time-line,
1512 * {@link LocalDateTime#truncatedTo(TemporalUnit) truncating}
1513 * the underlying local date-time. This is then converted back to a
1514 * {@code ZonedDateTime}, using the zone ID to obtain the offset.
1515 * <p>
1516 * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1517 * then the offset will be retained if possible, otherwise the earlier offset will be used.
1518 * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1519 * <p>
1520 * This instance is immutable and unaffected by this method call.
1521 *
1522 * @param unit the unit to truncate to, not null
1523 * @return a {@code ZonedDateTime} based on this date-time with the time truncated, not null
1524 * @throws DateTimeException if unable to truncate
1525 * @throws UnsupportedTemporalTypeException if the unit is not supported
1526 */
1527 public ZonedDateTime truncatedTo(TemporalUnit unit) {
1528     return resolveLocal(dateTime.truncatedTo(unit));
1529 }

```



```

1530 //-----
1531 /**
1532  * Returns a copy of this date-time with the specified amount added.
1533  * <p>
1534  * This returns a {@code ZonedDateTime}, based on this one, with the specified amount added.
1535  * The amount is typically {@link Period} or {@link Duration} but may be
1536  * any other type implementing the {@link TemporalAmount} interface.
1537  * <p>
1538  * The calculation is delegated to the amount object by calling
1539  * {@link TemporalAmount#addTo(Temporal)}. The amount implementation is free
1540  * to implement the addition in any way it wishes, however it typically
1541  * calls back to {@link #plus(long, TemporalUnit)}. Consult the documentation
1542  * of the amount implementation to determine if it can be successfully added.
1543  * <p>
1544  * This instance is immutable and unaffected by this method call.
1545  *
1546  * @param amountToAdd the amount to add, not null
1547  * @return a {@code ZonedDateTime} based on this date-time with the addition made, not null
1548  * @throws DateTimeException if the addition cannot be made
1549  * @throws ArithmeticException if numeric overflow occurs
1550  */
1551 @Override
1552 public ZonedDateTime plus(TemporalAmount amountToAdd) {
1553     if (amountToAdd instanceof Period) {
1554         Period periodToAdd = (Period) amountToAdd;
1555         return resolveLocal(dateTime.plus(periodToAdd));
1556     }
1557     Objects.requireNonNull(amountToAdd, "amountToAdd");
1558     return (ZonedDateTime) amountToAdd.addTo(this);
1559 }
1560
1561 /**
1562  * Returns a copy of this date-time with the specified amount added.
1563  * <p>
1564  * This returns a {@code ZonedDateTime}, based on this one, with the amount
1565  * in terms of the unit added. If it is not possible to add the amount, because the
1566  * unit is not supported or for some other reason, an exception is thrown.
1567  * <p>
1568  * If the field is a {@link ChronoUnit} then the addition is implemented here.
1569  * The zone is not part of the calculation and will be unchanged in the result.
1570  * The calculation for date and time units differ.
1571  * <p>
1572  * Date units operate on the local time-line.
1573  * The period is first added to the local date-time, then converted back
1574  * to a zoned date-time using the zone ID.
1575  * The conversion uses {@link #ofLocal(LocalDate, ZoneId, ZoneOffset)}
1576  * with the offset before the addition.
1577  * <p>
1578  * Time units operate on the instant time-line.
1579  * The period is first added to the local date-time, then converted back to
1580  * a zoned date-time using the zone ID.
1581  * The conversion uses {@link #ofInstant(LocalDate, ZoneOffset, ZoneId)}
1582  * with the offset before the addition.

```

```

1583 * <p>
1584 * If the field is not a {@code ChronoUnit}, then the result of this method
1585 * is obtained by invoking {@code TemporalUnit.addTo(Temporal, long)}
1586 * passing {@code this} as the argument. In this case, the unit determines
1587 * whether and how to perform the addition.
1588 * <p>
1589 * This instance is immutable and unaffected by this method call.
1590 *
1591 * @param amountToAdd the amount of the unit to add to the result, may be negative
1592 * @param unit the unit of the amount to add, not null
1593 * @return a {@code ZonedDateTime} based on this date-time with the specified amount added, not
1594 *         null
1595 * @throws DateTimeException if the addition cannot be made
1596 * @throws UnsupportedTemporalTypeException if the unit is not supported
1597 * @throws ArithmeticException if numeric overflow occurs
1598 */
1599 @Override
1600 public ZonedDateTime plus(long amountToAdd, TemporalUnit unit) {
1601     if (unit instanceof ChronoUnit) {
1602         if (unit.isDateBased()) {
1603             return resolveLocal(dateTime.plus(amountToAdd, unit));
1604         } else {
1605             return resolveInstant(dateTime.plus(amountToAdd, unit));
1606         }
1607     }
1608     return unit.addTo(this, amountToAdd);
1609 }
1610
1611 //-----
1612 /**
1613  * Returns a copy of this {@code ZonedDateTime} with the specified number of years added.
1614  * <p>
1615  * This operates on the local time-line,
1616  * {@link LocalDateTime#plusYears(long) adding years} to the local date-time.
1617  * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1618  * to obtain the offset.
1619  * <p>
1620  * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1621  * then the offset will be retained if possible, otherwise the earlier offset will be used.
1622  * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1623  * <p>
1624  * This instance is immutable and unaffected by this method call.
1625  *
1626  * @param years the years to add, may be negative
1627  * @return a {@code ZonedDateTime} based on this date-time with the years added, not null
1628  * @throws DateTimeException if the result exceeds the supported date range
1629  */
1630 public ZonedDateTime plusYears(long years) {
1631     return resolveLocal(dateTime.plusYears(years));
1632 }
1633
1634 /**
1635  * Returns a copy of this {@code ZonedDateTime} with the specified number of months added.

```

```

1635 * <p>
1636 * This operates on the local time-line,
1637 * {@link LocalDateTime#plusMonths(long) adding months} to the local date-time.
1638 * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1639 * to obtain the offset.
1640 * <p>
1641 * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1642 * then the offset will be retained if possible, otherwise the earlier offset will be used.
1643 * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1644 * <p>
1645 * This instance is immutable and unaffected by this method call.
1646 *
1647 * @param months the months to add, may be negative
1648 * @return a {@code ZonedDateTime} based on this date-time with the months added, not null
1649 * @throws DateTimeException if the result exceeds the supported date range
1650 */
1651 public ZonedDateTime plusMonths(long months) {
1652     return resolveLocal(dateTime.plusMonths(months));
1653 }
1654
1655 /**
1656 * Returns a copy of this {@code ZonedDateTime} with the specified number of weeks added.
1657 * <p>
1658 * This operates on the local time-line,
1659 * {@link LocalDateTime#plusWeeks(long) adding weeks} to the local date-time.
1660 * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1661 * to obtain the offset.
1662 * <p>
1663 * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1664 * then the offset will be retained if possible, otherwise the earlier offset will be used.
1665 * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1666 * <p>
1667 * This instance is immutable and unaffected by this method call.
1668 *
1669 * @param weeks the weeks to add, may be negative
1670 * @return a {@code ZonedDateTime} based on this date-time with the weeks added, not null
1671 * @throws DateTimeException if the result exceeds the supported date range
1672 */
1673 public ZonedDateTime plusWeeks(long weeks) {
1674     return resolveLocal(dateTime.plusWeeks(weeks));
1675 }
1676
1677 /**
1678 * Returns a copy of this {@code ZonedDateTime} with the specified number of days added.
1679 * <p>
1680 * This operates on the local time-line,
1681 * {@link LocalDateTime#plusDays(long) adding days} to the local date-time.
1682 * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1683 * to obtain the offset.
1684 * <p>
1685 * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1686 * then the offset will be retained if possible, otherwise the earlier offset will be used.
1687 * If in a gap, the local date-time will be adjusted forward by the length of the gap.

```

```

1688 * <p>
1689 * This instance is immutable and unaffected by this method call.
1690 *
1691 * @param days the days to add, may be negative
1692 * @return a {@code ZonedDateTime} based on this date-time with the days added, not null
1693 * @throws DateTimeException if the result exceeds the supported date range
1694 */
1695 public ZonedDateTime plusDays(long days) {
1696     return resolveLocal(dateTime.plusDays(days));
1697 }
1698
1699 //-----
1700 /**
1701 * Returns a copy of this {@code ZonedDateTime} with the specified number of hours added.
1702 * <p>
1703 * This operates on the instant time-line, such that adding one hour will
1704 * always be a duration of one hour later.
1705 * This may cause the local date-time to change by an amount other than one hour.
1706 * Note that this is a different approach to that used by days, months and years,
1707 * thus adding one day is not the same as adding 24 hours.
1708 * <p>
1709 * For example, consider a time-zone where the spring DST cutover means that the
1710 * local times 01:00 to 01:59 occur twice changing from offset +02:00 to +01:00.
1711 * <ul>
1712 * <li>Adding one hour to 00:30+02:00 will result in 01:30+02:00
1713 * <li>Adding one hour to 01:30+02:00 will result in 01:30+01:00
1714 * <li>Adding one hour to 01:30+01:00 will result in 02:30+01:00
1715 * <li>Adding three hours to 00:30+02:00 will result in 02:30+01:00
1716 * </ul>
1717 * <p>
1718 * This instance is immutable and unaffected by this method call.
1719 *
1720 * @param hours the hours to add, may be negative
1721 * @return a {@code ZonedDateTime} based on this date-time with the hours added, not null
1722 * @throws DateTimeException if the result exceeds the supported date range
1723 */
1724 public ZonedDateTime plusHours(long hours) {
1725     return resolveInstant(dateTime.plusHours(hours));
1726 }
1727
1728 /**
1729 * Returns a copy of this {@code ZonedDateTime} with the specified number of minutes added.
1730 * <p>
1731 * This operates on the instant time-line, such that adding one minute will
1732 * always be a duration of one minute later.
1733 * This may cause the local date-time to change by an amount other than one minute.
1734 * Note that this is a different approach to that used by days, months and years.
1735 * <p>
1736 * This instance is immutable and unaffected by this method call.
1737 *
1738 * @param minutes the minutes to add, may be negative
1739 * @return a {@code ZonedDateTime} based on this date-time with the minutes added, not null
1740 * @throws DateTimeException if the result exceeds the supported date range

```

```

1741     */
1742     public ZonedDateTime plusMinutes(long minutes) {
1743         return resolveInstant(dateTime.plusMinutes(minutes));
1744     }
1745
1746     /**
1747      * Returns a copy of this {@code ZonedDateTime} with the specified number of seconds added.
1748      * <p>
1749      * This operates on the instant time-line, such that adding one second will
1750      * always be a duration of one second later.
1751      * This may cause the local date-time to change by an amount other than one second.
1752      * Note that this is a different approach to that used by days, months and years.
1753      * <p>
1754      * This instance is immutable and unaffected by this method call.
1755      *
1756      * @param seconds the seconds to add, may be negative
1757      * @return a {@code ZonedDateTime} based on this date-time with the seconds added, not null
1758      * @throws DateTimeException if the result exceeds the supported date range
1759      */
1760     public ZonedDateTime plusSeconds(long seconds) {
1761         return resolveInstant(dateTime.plusSeconds(seconds));
1762     }
1763
1764     /**
1765      * Returns a copy of this {@code ZonedDateTime} with the specified number of nanoseconds added.
1766      * <p>
1767      * This operates on the instant time-line, such that adding one nano will
1768      * always be a duration of one nano later.
1769      * This may cause the local date-time to change by an amount other than one nano.
1770      * Note that this is a different approach to that used by days, months and years.
1771      * <p>
1772      * This instance is immutable and unaffected by this method call.
1773      *
1774      * @param nanos the nanos to add, may be negative
1775      * @return a {@code ZonedDateTime} based on this date-time with the nanoseconds added, not null
1776      * @throws DateTimeException if the result exceeds the supported date range
1777      */
1778     public ZonedDateTime plusNanos(long nanos) {
1779         return resolveInstant(dateTime.plusNanos(nanos));
1780     }
1781
1782     //-----
1783     /**
1784      * Returns a copy of this date-time with the specified amount subtracted.
1785      * <p>
1786      * This returns a {@code ZonedDateTime}, based on this one, with the specified amount
1787      * subtracted.
1788      * The amount is typically {@link Period} or {@link Duration} but may be
1789      * any other type implementing the {@link TemporalAmount} interface.
1790      * <p>
1791      * The calculation is delegated to the amount object by calling
1792      * {@link TemporalAmount#subtractFrom(Temporal)}. The amount implementation is free
1793      * to implement the subtraction in any way it wishes, however it typically

```

```

1793     * calls back to {@link #minus(long, TemporalUnit)}. Consult the documentation
1794     * of the amount implementation to determine if it can be successfully subtracted.
1795     * <p>
1796     * This instance is immutable and unaffected by this method call.
1797     *
1798     * @param amountToSubtract the amount to subtract, not null
1799     * @return a {@code ZonedDateTime} based on this date-time with the subtraction made, not null
1800     * @throws DateTimeException if the subtraction cannot be made
1801     * @throws ArithmeticException if numeric overflow occurs
1802     */
1803     @Override
1804     public ZonedDateTime minus(TemporalAmount amountToSubtract) {
1805         if (amountToSubtract instanceof Period) {
1806             Period periodToSubtract = (Period) amountToSubtract;
1807             return resolveLocal(dateTime.minus(periodToSubtract));
1808         }
1809         Objects.requireNonNull(amountToSubtract, "amountToSubtract");
1810         return (ZonedDateTime) amountToSubtract.subtractFrom(this);
1811     }
1812
1813     /**
1814     * Returns a copy of this date-time with the specified amount subtracted.
1815     * <p>
1816     * This returns a {@code ZonedDateTime}, based on this one, with the amount
1817     * in terms of the unit subtracted. If it is not possible to subtract the amount,
1818     * because the unit is not supported or for some other reason, an exception is thrown.
1819     * <p>
1820     * The calculation for date and time units differ.
1821     * <p>
1822     * Date units operate on the local time-line.
1823     * The period is first subtracted from the local date-time, then converted back
1824     * to a zoned date-time using the zone ID.
1825     * The conversion uses {@link #ofLocal(LocalDateTime, ZoneId, ZoneOffset)}
1826     * with the offset before the subtraction.
1827     * <p>
1828     * Time units operate on the instant time-line.
1829     * The period is first subtracted from the local date-time, then converted back to
1830     * a zoned date-time using the zone ID.
1831     * The conversion uses {@link #ofInstant(LocalDateTime, ZoneOffset, ZoneId)}
1832     * with the offset before the subtraction.
1833     * <p>
1834     * This method is equivalent to {@link #plus(long, TemporalUnit)} with the amount negated.
1835     * See that method for a full description of how addition, and thus subtraction, works.
1836     * <p>
1837     * This instance is immutable and unaffected by this method call.
1838     *
1839     * @param amountToSubtract the amount of the unit to subtract from the result, may be negative
1840     * @param unit the unit of the amount to subtract, not null
1841     * @return a {@code ZonedDateTime} based on this date-time with the specified amount subtracted
1842     *         , not null
1843     * @throws DateTimeException if the subtraction cannot be made
1844     * @throws UnsupportedTemporalTypeException if the unit is not supported
1845     * @throws ArithmeticException if numeric overflow occurs

```

```

1845     */
1846     @Override
1847     public ZonedDateTime minus(long amountToSubtract, TemporalUnit unit) {
1848         return (amountToSubtract == Long.MIN_VALUE ? plus(Long.MAX_VALUE, unit).plus(1, unit) :
1849             plus(-amountToSubtract, unit));
1850     }
1851
1852     //-----
1853     /**
1854      * Returns a copy of this {@code ZonedDateTime} with the specified number of years subtracted.
1855      * <p>
1856      * This operates on the local time-line,
1857      * {@link LocalDateTime#minusYears(long)} subtracting years} to the local date-time.
1858      * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1859      * to obtain the offset.
1860      * <p>
1861      * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1862      * then the offset will be retained if possible, otherwise the earlier offset will be used.
1863      * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1864      * <p>
1865      * This instance is immutable and unaffected by this method call.
1866      *
1867      * @param years the years to subtract, may be negative
1868      * @return a {@code ZonedDateTime} based on this date-time with the years subtracted, not null
1869      * @throws DateTimeException if the result exceeds the supported date range
1870      */
1871     public ZonedDateTime minusYears(long years) {
1872         return (years == Long.MIN_VALUE ? plusYears(Long.MAX_VALUE).plusYears(1) : plusYears(-years
1873             ));
1874     }
1875
1876     /**
1877      * Returns a copy of this {@code ZonedDateTime} with the specified number of months subtracted.
1878      * <p>
1879      * This operates on the local time-line,
1880      * {@link LocalDateTime#minusMonths(long)} subtracting months} to the local date-time.
1881      * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1882      * to obtain the offset.
1883      * <p>
1884      * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1885      * then the offset will be retained if possible, otherwise the earlier offset will be used.
1886      * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1887      * <p>
1888      * This instance is immutable and unaffected by this method call.
1889      *
1890      * @param months the months to subtract, may be negative
1891      * @return a {@code ZonedDateTime} based on this date-time with the months subtracted, not null
1892      * @throws DateTimeException if the result exceeds the supported date range
1893      */
1894     public ZonedDateTime minusMonths(long months) {
1895         return (months == Long.MIN_VALUE ? plusMonths(Long.MAX_VALUE).plusMonths(1) : plusMonths(-
1896             months));
1897     }

```



```

1895
1896 /**
1897  * Returns a copy of this {@code ZonedDateTime} with the specified number of weeks subtracted.
1898  * <p>
1899  * This operates on the local time-line,
1900  * {@link LocalDateTime#minusWeeks(long) subtracting weeks} to the local date-time.
1901  * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1902  * to obtain the offset.
1903  * <p>
1904  * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1905  * then the offset will be retained if possible, otherwise the earlier offset will be used.
1906  * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1907  * <p>
1908  * This instance is immutable and unaffected by this method call.
1909  *
1910  * @param weeks the weeks to subtract, may be negative
1911  * @return a {@code ZonedDateTime} based on this date-time with the weeks subtracted, not null
1912  * @throws DateTimeException if the result exceeds the supported date range
1913  */
1914 public ZonedDateTime minusWeeks(long weeks) {
1915     return (weeks == Long.MIN_VALUE ? plusWeeks(Long.MAX_VALUE).plusWeeks(1) : plusWeeks(-weeks
1916     ));
1917 }
1918
1919 /**
1920  * Returns a copy of this {@code ZonedDateTime} with the specified number of days subtracted.
1921  * <p>
1922  * This operates on the local time-line,
1923  * {@link LocalDateTime#minusDays(long) subtracting days} to the local date-time.
1924  * This is then converted back to a {@code ZonedDateTime}, using the zone ID
1925  * to obtain the offset.
1926  * <p>
1927  * When converting back to {@code ZonedDateTime}, if the local date-time is in an overlap,
1928  * then the offset will be retained if possible, otherwise the earlier offset will be used.
1929  * If in a gap, the local date-time will be adjusted forward by the length of the gap.
1930  * <p>
1931  * This instance is immutable and unaffected by this method call.
1932  *
1933  * @param days the days to subtract, may be negative
1934  * @return a {@code ZonedDateTime} based on this date-time with the days subtracted, not null
1935  * @throws DateTimeException if the result exceeds the supported date range
1936  */
1937 public ZonedDateTime minusDays(long days) {
1938     return (days == Long.MIN_VALUE ? plusDays(Long.MAX_VALUE).plusDays(1) : plusDays(-days));
1939 }
1940
1941 //-----
1942 /**
1943  * Returns a copy of this {@code ZonedDateTime} with the specified number of hours subtracted.
1944  * <p>
1945  * This operates on the instant time-line, such that subtracting one hour will
1946  * always be a duration of one hour earlier.
1947  * This may cause the local date-time to change by an amount other than one hour.

```



```

1947     * Note that this is a different approach to that used by days, months and years,
1948     * thus subtracting one day is not the same as adding 24 hours.
1949     * <p>
1950     * For example, consider a time-zone where the spring DST cutover means that the
1951     * local times 01:00 to 01:59 occur twice changing from offset +02:00 to +01:00.
1952     * <ul>
1953     * <li>Subtracting one hour from 02:30+01:00 will result in 01:30+02:00
1954     * <li>Subtracting one hour from 01:30+01:00 will result in 01:30+02:00
1955     * <li>Subtracting one hour from 01:30+02:00 will result in 00:30+01:00
1956     * <li>Subtracting three hours from 02:30+01:00 will result in 00:30+02:00
1957     * </ul>
1958     * <p>
1959     * This instance is immutable and unaffected by this method call.
1960     *
1961     * @param hours    the hours to subtract, may be negative
1962     * @return a {@code ZonedDateTime} based on this date-time with the hours subtracted, not null
1963     * @throws DateTimeException if the result exceeds the supported date range
1964     */
1965     public ZonedDateTime minusHours(long hours) {
1966         return (hours == Long.MIN_VALUE ? plusHours(Long.MAX_VALUE).plusHours(1) : plusHours(-hours
1967         ));
1968     }
1969
1970     /**
1971     * Returns a copy of this {@code ZonedDateTime} with the specified number of minutes subtracted
1972     * .
1973     * <p>
1974     * This operates on the instant time-line, such that subtracting one minute will
1975     * always be a duration of one minute earlier.
1976     * This may cause the local date-time to change by an amount other than one minute.
1977     * Note that this is a different approach to that used by days, months and years.
1978     * <p>
1979     * This instance is immutable and unaffected by this method call.
1980     *
1981     * @param minutes    the minutes to subtract, may be negative
1982     * @return a {@code ZonedDateTime} based on this date-time with the minutes subtracted, not
1983     *         null
1984     * @throws DateTimeException if the result exceeds the supported date range
1985     */
1986     public ZonedDateTime minusMinutes(long minutes) {
1987         return (minutes == Long.MIN_VALUE ? plusMinutes(Long.MAX_VALUE).plusMinutes(1) :
1988         plusMinutes(-minutes));
1989     }
1990
1991     /**
1992     * Returns a copy of this {@code ZonedDateTime} with the specified number of seconds subtracted
1993     * .
1994     * <p>
1995     * This operates on the instant time-line, such that subtracting one second will
1996     * always be a duration of one second earlier.
1997     * This may cause the local date-time to change by an amount other than one second.
1998     * Note that this is a different approach to that used by days, months and years.
1999     * <p>

```

```

1995     * This instance is immutable and unaffected by this method call.
1996     *
1997     * @param seconds the seconds to subtract, may be negative
1998     * @return a {@code ZonedDateTime} based on this date-time with the seconds subtracted, not
1999     *         null
2000     * @throws DateTimeException if the result exceeds the supported date range
2001     */
2002     public ZonedDateTime minusSeconds(long seconds) {
2003         return (seconds == Long.MIN_VALUE ? plusSeconds(Long.MAX_VALUE).plusSeconds(1) :
2004             plusSeconds(-seconds));
2005     }
2006
2007     /**
2008     * Returns a copy of this {@code ZonedDateTime} with the specified number of nanoseconds
2009     * subtracted.
2010     *
2011     * <p>
2012     * This operates on the instant time-line, such that subtracting one nano will
2013     * always be a duration of one nano earlier.
2014     * This may cause the local date-time to change by an amount other than one nano.
2015     * Note that this is a different approach to that used by days, months and years.
2016     * <p>
2017     * This instance is immutable and unaffected by this method call.
2018     *
2019     * @param nanos the nanos to subtract, may be negative
2020     * @return a {@code ZonedDateTime} based on this date-time with the nanoseconds subtracted, not
2021     *         null
2022     * @throws DateTimeException if the result exceeds the supported date range
2023     */
2024     public ZonedDateTime minusNanos(long nanos) {
2025         return (nanos == Long.MIN_VALUE ? plusNanos(Long.MAX_VALUE).plusNanos(1) : plusNanos(-nanos
2026             ));
2027     }
2028
2029     //-----
2030     /**
2031     * Queries this date-time using the specified query.
2032     *
2033     * <p>
2034     * This queries this date-time using the specified query strategy object.
2035     * The {@code TemporalQuery} object defines the logic to be used to
2036     * obtain the result. Read the documentation of the query to understand
2037     * what the result of this method will be.
2038     * <p>
2039     * The result of this method is obtained by invoking the
2040     * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
2041     * specified query passing {@code this} as the argument.
2042     *
2043     * @param <R> the type of the result
2044     * @param query the query to invoke, not null
2045     * @return the query result, null may be returned (defined by the query)
2046     * @throws DateTimeException if unable to query (defined by the query)
2047     * @throws ArithmeticException if numeric overflow occurs (defined by the query)
2048     */
2049     @SuppressWarnings("unchecked")

```

```

2043 @Override // override for Javadoc
2044 public <R> R query(TemporalQuery<R> query) {
2045     if (query == TemporalQueries.localDate()) {
2046         return (R) toLocalDate();
2047     }
2048     return ChronoZonedDateTime.super.query(query);
2049 }
2050
2051 /**
2052  * Calculates the amount of time until another date-time in terms of the specified unit.
2053  * <p>
2054  * This calculates the amount of time between two {@code ZonedDateTime}
2055  * objects in terms of a single {@code TemporalUnit}.
2056  * The start and end points are {@code this} and the specified date-time.
2057  * The result will be negative if the end is before the start.
2058  * For example, the amount in days between two date-times can be calculated
2059  * using {@code startDateTime.until(endDateTime, DAYS)}.
2060  * <p>
2061  * The {@code Temporal} passed to this method is converted to a
2062  * {@code ZonedDateTime} using {@code #from(TemporalAccessor)}.
2063  * If the time-zone differs between the two zoned date-times, the specified
2064  * end date-time is normalized to have the same zone as this date-time.
2065  * <p>
2066  * The calculation returns a whole number, representing the number of
2067  * complete units between the two date-times.
2068  * For example, the amount in months between 2012-06-15T00:00Z and 2012-08-14T23:59Z
2069  * will only be one month as it is one minute short of two months.
2070  * <p>
2071  * There are two equivalent ways of using this method.
2072  * The first is to invoke this method.
2073  * The second is to use {@code TemporalUnit#between(Temporal, Temporal)}:
2074  * <pre>
2075  *     // these two lines are equivalent
2076  *     amount = start.until(end, MONTHS);
2077  *     amount = MONTHS.between(start, end);
2078  * </pre>
2079  * The choice should be made based on which makes the code more readable.
2080  * <p>
2081  * The calculation is implemented in this method for {@code ChronoUnit}.
2082  * The units {@code NANOS}, {@code MICROS}, {@code MILLIS}, {@code SECONDS},
2083  * {@code MINUTES}, {@code HOURS} and {@code HALF_DAYS}, {@code DAYS},
2084  * {@code WEEKS}, {@code MONTHS}, {@code YEARS}, {@code DECADES},
2085  * {@code CENTURIES}, {@code MILLENNIA} and {@code ERAS} are supported.
2086  * Other {@code ChronoUnit} values will throw an exception.
2087  * <p>
2088  * The calculation for date and time units differ.
2089  * <p>
2090  * Date units operate on the local time-line, using the local date-time.
2091  * For example, the period from noon on day 1 to noon the following day
2092  * in days will always be counted as exactly one day, irrespective of whether
2093  * there was a daylight savings change or not.
2094  * <p>
2095  * Time units operate on the instant time-line.

```

```

2096     * The calculation effectively converts both zoned date-times to instants
2097     * and then calculates the period between the instants.
2098     * For example, the period from noon on day 1 to noon the following day
2099     * in hours may be 23, 24 or 25 hours (or some other amount) depending on
2100     * whether there was a daylight savings change or not.
2101     * <p>
2102     * If the unit is not a {@code ChronoUnit}, then the result of this method
2103     * is obtained by invoking {@code TemporalUnit.between(Temporal, Temporal)}
2104     * passing {@code this} as the first argument and the converted input temporal
2105     * as the second argument.
2106     * <p>
2107     * This instance is immutable and unaffected by this method call.
2108     *
2109     * @param endExclusive the end date, exclusive, which is converted to a {@code ZonedDateTime},
2110         not null
2111     * @param unit the unit to measure the amount in, not null
2112     * @return the amount of time between this date-time and the end date-time
2113     * @throws DateTimeException if the amount cannot be calculated, or the end
2114     *         temporal cannot be converted to a {@code ZonedDateTime}
2115     * @throws UnsupportedTemporalTypeException if the unit is not supported
2116     * @throws ArithmeticException if numeric overflow occurs
2117     */
2118     @Override
2119     public long until(Temporal endExclusive, TemporalUnit unit) {
2120         ZonedDateTime end = ZonedDateTime.from(endExclusive);
2121         if (unit instanceof ChronoUnit) {
2122             end = end.withZoneSameInstant(zone);
2123             if (unit.isDateBased()) {
2124                 return dateTime.until(end.dateTime, unit);
2125             } else {
2126                 return toOffsetDateTime().until(end.toOffsetDateTime(), unit);
2127             }
2128         }
2129         return unit.between(this, end);
2130     }
2131
2132     /**
2133     * Formats this date-time using the specified formatter.
2134     * <p>
2135     * This date-time will be passed to the formatter to produce a string.
2136     *
2137     * @param formatter the formatter to use, not null
2138     * @return the formatted date-time string, not null
2139     * @throws DateTimeException if an error occurs during printing
2140     */
2141     @Override // override for Javadoc and performance
2142     public String format(DateTimeFormatter formatter) {
2143         Objects.requireNonNull(formatter, "formatter");
2144         return formatter.format(this);
2145     }
2146
2147     //-----
2148     /**

```

```

2148     * Converts this date-time to an {@code OffsetDateTime}.
2149     * <p>
2150     * This creates an offset date-time using the local date-time and offset.
2151     * The zone ID is ignored.
2152     *
2153     * @return an offset date-time representing the same local date-time and offset, not null
2154     */
2155     public OffsetDateTime toOffsetDateTime() {
2156         return OffsetDateTime.of(dateTime, offset);
2157     }
2158
2159     //-----
2160     /**
2161     * Checks if this date-time is equal to another date-time.
2162     * <p>
2163     * The comparison is based on the offset date-time and the zone.
2164     * Only objects of type {@code ZonedDateTime} are compared, other types return false.
2165     *
2166     * @param obj the object to check, null returns false
2167     * @return true if this is equal to the other date-time
2168     */
2169     @Override
2170     public boolean equals(Object obj) {
2171         if (this == obj) {
2172             return true;
2173         }
2174         if (obj instanceof ZonedDateTime) {
2175             ZonedDateTime other = (ZonedDateTime) obj;
2176             return dateTime.equals(other.dateTime) &&
2177                 offset.equals(other.offset) &&
2178                 zone.equals(other.zone);
2179         }
2180         return false;
2181     }
2182
2183     /**
2184     * A hash code for this date-time.
2185     *
2186     * @return a suitable hash code
2187     */
2188     @Override
2189     public int hashCode() {
2190         return dateTime.hashCode() ^ offset.hashCode() ^ Integer.rotateLeft(zone.hashCode(), 3);
2191     }
2192
2193     //-----
2194     /**
2195     * Outputs this date-time as a {@code String}, such as
2196     * {@code 2007-12-03T10:15:30+01:00[Europe/Paris]}.
2197     * <p>
2198     * The format consists of the {@code LocalDateTime} followed by the {@code ZoneOffset}.
2199     * If the {@code ZoneId} is not the same as the offset, then the ID is output.
2200     * The output is compatible with ISO-8601 if the offset and ID are the same.

```

```

2201     *
2202     * @return a string representation of this date-time, not null
2203     */
2204     @Override // override for Javadoc
2205     public String toString() {
2206         String str = dateTime.toString() + offset.toString();
2207         if (offset != zone) {
2208             str += '[' + zone.toString() + ']';
2209         }
2210         return str;
2211     }
2212
2213     //-----
2214     /**
2215      * Writes the object using a
2216      * <a href=".../serialized-form.html#java.time.Ser">dedicated serialized form</a>.
2217      * @serialData
2218      * <pre>
2219      *   out.writeByte(6); // identifies a ZonedDateTime
2220      *   // the <a href=".../serialized-form.html#java.time.LocalDateTime">dateTime</a> excluding
2221      *     the one byte header
2222      *   // the <a href=".../serialized-form.html#java.time.ZoneOffset">offset</a> excluding the
2223      *     one byte header
2224      *   // the <a href=".../serialized-form.html#java.time.ZoneId">zone ID</a> excluding the one
2225      *     byte header
2226      * </pre>
2227      *
2228      * @return the instance of {@code Ser}, not null
2229      */
2230     private Object writeReplace() {
2231         return new Ser(Ser.ZONE_DATE_TIME_TYPE, this);
2232     }
2233
2234     /**
2235      * Defend against malicious streams.
2236      *
2237      * @param s the stream to read
2238      * @throws InvalidObjectException always
2239      */
2240     private void readObject(ObjectInputStream s) throws InvalidObjectException {
2241         throw new InvalidObjectException("Deserialization via serialization delegate");
2242     }
2243
2244     void writeExternal(DataOutput out) throws IOException {
2245         dateTime.writeExternal(out);
2246         offset.writeExternal(out);
2247         zone.write(out);
2248     }
2249
2250     static ZonedDateTime readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
2251         LocalDateTime dateTime = LocalDateTime.readExternal(in);
2252         ZoneOffset offset = ZoneOffset.readExternal(in);
2253         ZoneId zone = (ZoneId) Ser.read(in);

```

```
2251     return ZonedDateTime.ofLenient(dateTime, offset, zone);
2252 }
2253
2254 }
```

4.21 package-info.java

Secuencia 70: java/time/package-info.java

```
1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
```

```

46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
62
63 /**
64 * <p>
65 * The main API for dates, times, instants, and durations.
66 * </p>
67 * <p>
68 * The classes defined here represent the principle date-time concepts,
69 * including instants, durations, dates, times, time-zones and periods.
70 * They are based on the ISO calendar system, which is the <i>de facto</i> world
71 * calendar following the proleptic Gregorian rules.
72 * All the classes are immutable and thread-safe.
73 * </p>
74 * <p>
75 * Each date time instance is composed of fields that are conveniently
76 * made available by the APIs. For lower level access to the fields refer
77 * to the {@code java.time.temporal} package.
78 * Each class includes support for printing and parsing all manner of dates and times.
79 * Refer to the {@code java.time.format} package for customization options.
80 * </p>
81 * <p>
82 * The {@code java.time.chrono} package contains the calendar neutral API
83 * {@link java.time.chrono.ChronoLocalDate ChronoLocalDate},
84 * {@link java.time.chrono.ChronoLocalDateTime ChronoLocalDateTime},
85 * {@link java.time.chrono.ChronoZonedDateTime ChronoZonedDateTime} and
86 * {@link java.time.chrono.Era Era}.
87 * This is intended for use by applications that need to use localized calendars.
88 * It is recommended that applications use the ISO-8601 date and time classes from
89 * this package across system boundaries, such as to the database or across the network.
90 * The calendar neutral API should be reserved for interactions with users.
91 * </p>
92 *
93 * <h3>Dates and Times</h3>
94 * <p>
95 * {@link java.time.Instant} is essentially a numeric timestamp.
96 * The current Instant can be retrieved from a {@link java.time.Clock}.
97 * This is useful for logging and persistence of a point in time
98 * and has in the past been associated with storing the result

```



```
99  * from {@link java.lang.System#currentTimeMillis()}.
100 * </p>
101 * <p>
102 * {@link java.time.LocalDate} stores a date without a time.
103 * This stores a date like '2010-12-03' and could be used to store a birthday.
104 * </p>
105 * <p>
106 * {@link java.time.LocalTime} stores a time without a date.
107 * This stores a time like '11:30' and could be used to store an opening or closing time.
108 * </p>
109 * <p>
110 * {@link java.time.LocalDateTime} stores a date and time.
111 * This stores a date-time like '2010-12-03T11:30'.
112 * </p>
113 * <p>
114 * {@link java.time.ZonedDateTime} stores a date and time with a time-zone.
115 * This is useful if you want to perform accurate calculations of
116 * dates and times taking into account the {@link java.time.ZoneId}, such as 'Europe/Paris'.
117 * Where possible, it is recommended to use a simpler class without a time-zone.
118 * The widespread use of time-zones tends to add considerable complexity to an application.
119 * </p>
120 *
121 * <h3>Duration and Period</h3>
122 * <p>
123 * Beyond dates and times, the API also allows the storage of periods and durations of time.
124 * A {@link java.time.Duration} is a simple measure of time along the time-line in nanoseconds.
125 * A {@link java.time.Period} expresses an amount of time in units meaningful
126 * to humans, such as years or days.
127 * </p>
128 *
129 * <h3>Additional value types</h3>
130 * <p>
131 * {@link java.time.Month} stores a month on its own.
132 * This stores a single month-of-year in isolation, such as 'DECEMBER'.
133 * </p>
134 * <p>
135 * {@link java.time.DayOfWeek} stores a day-of-week on its own.
136 * This stores a single day-of-week in isolation, such as 'TUESDAY'.
137 * </p>
138 * <p>
139 * {@link java.time.Year} stores a year on its own.
140 * This stores a single year in isolation, such as '2010'.
141 * </p>
142 * <p>
143 * {@link java.time.YearMonth} stores a year and month without a day or time.
144 * This stores a year and month, such as '2010-12' and could be used for a credit card expiry.
145 * </p>
146 * <p>
147 * {@link java.time.MonthDay} stores a month and day without a year or time.
148 * This stores a month and day-of-month, such as '--12-03' and
149 * could be used to store an annual event like a birthday without storing the year.
150 * </p>
151 * <p>
```

```

152 * {@link java.time.OffsetTime} stores a time and offset from UTC without a date.
153 * This stores a date like '11:30+01:00'.
154 * The {@link java.time.ZoneOffset ZoneOffset} is of the form '+01:00'.
155 * </p>
156 * <p>
157 * {@link java.time.OffsetDateTime} stores a date and time and offset from UTC.
158 * This stores a date-time like '2010-12-03T11:30+01:00'.
159 * This is sometimes found in XML messages and other forms of persistence,
160 * but contains less information than a full time-zone.
161 * </p>
162 *
163 * <h3>Package specification</h3>
164 * <p>
165 * Unless otherwise noted, passing a null argument to a constructor or method in any class or
166 * interface
167 * in this package will cause a {@link java.lang.NullPointerException NullPointerException} to be
168 * thrown.
169 * The Javadoc "@param" definition is used to summarise the null-behavior.
170 * The "@throws {@link java.lang.NullPointerException}" is not explicitly documented in each method
171 * .
172 * </p>
173 * <p>
174 * All calculations should check for numeric overflow and throw either an {@link java.lang.
175 * ArithmeticException}
176 * or a {@link java.time.DateTimeException}.
177 * </p>
178 *
179 * <h3>Design notes (non normative)</h3>
180 * <p>
181 * The API has been designed to reject null early and to be clear about this behavior.
182 * A key exception is any method that takes an object and returns a boolean, for the purpose
183 * of checking or validating, will generally return false for null.
184 * </p>
185 * <p>
186 * The API is designed to be type-safe where reasonable in the main high-level API.
187 * Thus, there are separate classes for the distinct concepts of date, time and date-time,
188 * plus variants for offset and time-zone.
189 * This can seem like a lot of classes, but most applications can begin with just five date/time
190 * types.
191 * <ul>
192 * <li>{@link java.time.Instant} - a timestamp</li>
193 * <li>{@link java.time.LocalDate} - a date without a time, or any reference to an offset or time-
194 * zone</li>
195 * <li>{@link java.time.LocalTime} - a time without a date, or any reference to an offset or time-
196 * zone</li>
197 * <li>{@link java.time.LocalDateTime} - combines date and time, but still without any offset or
198 * time-zone</li>
199 * <li>{@link java.time.ZonedDateTime} - a "full" date-time with time-zone and resolved offset from
200 * UTC/Greenwich</li>
201 * </ul>
202 * <p>
203 * {@code Instant} is the closest equivalent class to {@code java.util.Date}.
204 * {@code ZonedDateTime} is the closest equivalent class to {@code java.util.GregorianCalendar}.

```

```
196 * </p>
197 * <p>
198 * Where possible, applications should use {@code LocalDate}, {@code LocalTime} and {@code
    LocalDateTime}
199 * to better model the domain. For example, a birthday should be stored in a code {@code LocalDate
    }.
200 * Bear in mind that any use of a {@linkplain java.time.ZoneId time-zone}, such as 'Europe/Paris',
    adds
201 * considerable complexity to a calculation.
202 * Many applications can be written only using {@code LocalDate}, {@code LocalTime} and {@code
    Instant},
203 * with the time-zone added at the user interface (UI) layer.
204 * </p>
205 * <p>
206 * The offset-based date-time types {@code OffsetTime} and {@code OffsetDateTime},
207 * are intended primarily for use with network protocols and database access.
208 * For example, most databases cannot automatically store a time-zone like 'Europe/Paris', but
209 * they can store an offset like '+02:00'.
210 * </p>
211 * <p>
212 * Classes are also provided for the most important sub-parts of a date, including {@code Month},
213 * {@code DayOfWeek}, {@code Year}, {@code YearMonth} and {@code MonthDay}.
214 * These can be used to model more complex date-time concepts.
215 * For example, {@code YearMonth} is useful for representing a credit card expiry.
216 * </p>
217 * <p>
218 * Note that while there are a large number of classes representing different aspects of dates,
219 * there are relatively few dealing with different aspects of time.
220 * Following type-safety to its logical conclusion would have resulted in classes for
221 * hour-minute, hour-minute-second and hour-minute-second-nanosecond.
222 * While logically pure, this was not a practical option as it would have almost tripled the
223 * number of classes due to the combinations of date and time.
224 * Thus, {@code LocalTime} is used for all precisions of time, with zeroes used to imply lower
    precision.
225 * </p>
226 * <p>
227 * Following full type-safety to its ultimate conclusion might also argue for a separate class
228 * for each field in date-time, such as a class for HourOfDay and another for DayOfMonth.
229 * This approach was tried, but was excessively complicated in the Java language, lacking usability
    .
230 * A similar problem occurs with periods.
231 * There is a case for a separate class for each period unit, such as a type for Years and a type
    for Minutes.
232 * However, this yields a lot of classes and a problem of type conversion.
233 * Thus, the set of date-time types provided is a compromise between purity and practicality.
234 * </p>
235 * <p>
236 * The API has a relatively large surface area in terms of number of methods.
237 * This is made manageable through the use of consistent method prefixes.
238 * <ul>
239 * <li>{@code of} - static factory method</li>
240 * <li>{@code parse} - static factory method focussed on parsing</li>
241 * <li>{@code get} - gets the value of something</li>
```

```

242 * <li>{@code is} - checks if something is true</li>
243 * <li>{@code with} - the immutable equivalent of a setter</li>
244 * <li>{@code plus} - adds an amount to an object</li>
245 * <li>{@code minus} - subtracts an amount from an object</li>
246 * <li>{@code to} - converts this object to another type</li>
247 * <li>{@code at} - combines this object with another, such as {@code date.atTime(time)}</li>
248 * </ul>
249 * <p>
250 * Multiple calendar systems is an awkward addition to the design challenges.
251 * The first principle is that most users want the standard ISO calendar system.
252 * As such, the main classes are ISO-only. The second principle is that most of those that want a
253 * non-ISO calendar system want it for user interaction, thus it is a UI localization issue.
254 * As such, date and time objects should be held as ISO objects in the data model and persistent
255 * storage, only being converted to and from a local calendar for display.
256 * The calendar system would be stored separately in the user preferences.
257 * </p>
258 * <p>
259 * There are, however, some limited use cases where users believe they need to store and use
260 * dates in arbitrary calendar systems throughout the application.
261 * This is supported by {@link java.time.chrono.ChronoLocalDate}, however it is vital to read
262 * all the associated warnings in the Javadoc of that interface before using it.
263 * In summary, applications that require general interoperation between multiple calendar systems
264 * typically need to be written in a very different way to those only using the ISO calendar,
265 * thus most applications should just use ISO and avoid {@code ChronoLocalDate}.
266 * </p>
267 * <p>
268 * The API is also designed for user extensibility, as there are many ways of calculating time.
269 * The {@linkplain java.time.temporal.TemporalField field} and {@linkplain java.time.temporal.
    TemporalUnit unit}
270 * API, accessed via {@link java.time.temporal.TemporalAccessor TemporalAccessor} and
271 * {@link java.time.temporal.Temporal Temporal} provide considerable flexibility to applications.
272 * In addition, the {@link java.time.temporal.TemporalQuery TemporalQuery} and
273 * {@link java.time.temporal.TemporalAdjuster TemporalAdjuster} interfaces provide day-to-day
274 * power, allowing code to read close to business requirements:
275 * </p>
276 * <pre>
277 *     LocalDate customerBirthday = customer.loadBirthdayFromDatabase();
278 *     LocalDate today = LocalDate.now();
279 *     if (customerBirthday.equals(today)) {
280 *         LocalDate specialOfferExpiryDate = today.plusWeeks(2).with(next(FRIDAY));
281 *         customer.sendBirthdaySpecialOffer(specialOfferExpiryDate);
282 *     }
283 *
284 * </pre>
285 *
286 * @since JDK1.8
287 */
288 package java.time;

```

5 Time Chrono (java.time.chrono package)

5.1 AbstractChronology.java

Secuencia 71: java/time/chrono/AbstractChronology.java

```
1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
```

```
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.chrono;
63
64  import static java.time.temporal.ChronoField.ALIGNED_DAY_OF_WEEK_IN_MONTH;
65  import static java.time.temporal.ChronoField.ALIGNED_DAY_OF_WEEK_IN_YEAR;
66  import static java.time.temporal.ChronoField.ALIGNED_WEEK_OF_MONTH;
67  import static java.time.temporal.ChronoField.ALIGNED_WEEK_OF_YEAR;
68  import static java.time.temporal.ChronoField.DAY_OF_MONTH;
69  import static java.time.temporal.ChronoField.DAY_OF_WEEK;
70  import static java.time.temporal.ChronoField.DAY_OF_YEAR;
71  import static java.time.temporal.ChronoField.EPOCH_DAY;
72  import static java.time.temporal.ChronoField.ERA;
73  import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
74  import static java.time.temporal.ChronoField.PROLEPTIC_MONTH;
75  import static java.time.temporal.ChronoField.YEAR;
76  import static java.time.temporal.ChronoField.YEAR_OF_ERA;
77  import static java.time.temporal.ChronoUnit.DAYS;
78  import static java.time.temporal.ChronoUnit.MONTHS;
79  import static java.time.temporal.ChronoUnit.WEEKS;
80  import static java.time.temporal.TemporalAdjusters.nextOrSame;
81
82  import java.io.DataInput;
83  import java.io.DataOutput;
84  import java.io.IOException;
85  import java.io.InvalidObjectException;
86  import java.io.ObjectInputStream;
87  import java.io.ObjectStreamException;
88  import java.io.Serializable;
89  import java.time.DateTimeException;
90  import java.time.DayOfWeek;
91  import java.time.format.ResolverStyle;
92  import java.time.temporal.ChronoField;
93  import java.time.temporal.TemporalAdjusters;
94  import java.time.temporal.TemporalField;
95  import java.time.temporal.ValueRange;
96  import java.util.Comparator;
97  import java.util.HashSet;
98  import java.util.List;
99  import java.util.Locale;
100 import java.util.Map;
101 import java.util.Objects;
102 import java.util.ServiceLoader;
103 import java.util.Set;
104 import java.util.concurrent.ConcurrentHashMap;
105
```

```

106 import sun.util.logging.PlatformLogger;
107
108 /**
109  * An abstract implementation of a calendar system, used to organize and identify dates.
110  * <p>
111  * The main date and time API is built on the ISO calendar system.
112  * The chronology operates behind the scenes to represent the general concept of a calendar system.
113  * <p>
114  * See {@link Chronology} for more details.
115  *
116  * @implSpec
117  * This class is separated from the {@code Chronology} interface so that the static methods
118  * are not inherited. While {@code Chronology} can be implemented directly, it is strongly
119  * recommended to extend this abstract class instead.
120  * <p>
121  * This class must be implemented with care to ensure other classes operate correctly.
122  * All implementations that can be instantiated must be final, immutable and thread-safe.
123  * Subclasses should be Serializable wherever possible.
124  *
125  * @since 1.8
126  */
127 public abstract class AbstractChronology implements Chronology {
128
129     /**
130      * ChronoLocalDate order constant.
131      */
132     static final Comparator<ChronoLocalDate> DATE_ORDER =
133         (Comparator<ChronoLocalDate> & Serializable) (date1, date2) -> {
134             return Long.compare(date1.toEpochDay(), date2.toEpochDay());
135         };
136
137     /**
138      * ChronoLocalDateTime order constant.
139      */
140     static final Comparator<ChronoLocalDateTime<? extends ChronoLocalDate>> DATE_TIME_ORDER =
141         (Comparator<ChronoLocalDateTime<? extends ChronoLocalDate>> & Serializable) (dateTime1,
142             dateTime2) -> {
143             int cmp = Long.compare(dateTime1.toLocalDate().toEpochDay(), dateTime2.toLocalDate().
144                 toEpochDay());
145             if (cmp == 0) {
146                 cmp = Long.compare(dateTime1.toLocalTime().toNanoOfDay(), dateTime2.toLocalTime().
147                     toNanoOfDay());
148             }
149             return cmp;
150         };
151
152     /**
153      * ChronoZonedDateTime order constant.
154      */
155     static final Comparator<ChronoZonedDateTime<?>> INSTANT_ORDER =
156         (Comparator<ChronoZonedDateTime<?>> & Serializable) (dateTime1, dateTime2) -> {
157             int cmp = Long.compare(dateTime1.toEpochSecond(), dateTime2.toEpochSecond());
158             if (cmp == 0) {
159                 cmp = Long.compare(dateTime1.toLocalTime().getNano(), dateTime2.toLocalTime().
160                     getNano());
161             }
162             return cmp;
163         };
164
165     /**
166      * ChronoInstant order constant.
167      */
168     static final Comparator<ChronoInstant> INSTANT_ORDER =
169         (Comparator<ChronoInstant> & Serializable) (instant1, instant2) -> {
170             return Long.compare(instant1.toEpochSecond(), instant2.toEpochSecond());
171         };
172
173     /**
174      * ChronoPeriod order constant.
175      */
176     static final Comparator<ChronoPeriod> PERIOD_ORDER =
177         (Comparator<ChronoPeriod> & Serializable) (period1, period2) -> {
178             return Long.compare(period1.getDays(), period2.getDays());
179         };
180
181     /**
182      * ChronoDuration order constant.
183      */
184     static final Comparator<ChronoDuration> DURATION_ORDER =
185         (Comparator<ChronoDuration> & Serializable) (duration1, duration2) -> {
186             return Long.compare(duration1.getSeconds(), duration2.getSeconds());
187         };
188
189     /**
190      * ChronoYear order constant.
191      */
192     static final Comparator<ChronoYear> YEAR_ORDER =
193         (Comparator<ChronoYear> & Serializable) (year1, year2) -> {
194             return Long.compare(year1.getYear(), year2.getYear());
195         };
196
197     /**
198      * ChronoYearMonth order constant.
199      */
200     static final Comparator<ChronoYearMonth> YEAR_MONTH_ORDER =
201         (Comparator<ChronoYearMonth> & Serializable) (yearMonth1, yearMonth2) -> {
202             return Long.compare(yearMonth1.getYear(), yearMonth2.getYear());
203         };
204
205     /**
206      * ChronoYearMonthDay order constant.
207      */
208     static final Comparator<ChronoYearMonthDay> YEAR_MONTH_DAY_ORDER =
209         (Comparator<ChronoYearMonthDay> & Serializable) (yearMonthDay1, yearMonthDay2) -> {
210             return Long.compare(yearMonthDay1.getYear(), yearMonthDay2.getYear());
211         };
212
213     /**
214      * ChronoYearMonthDayHour order constant.
215      */
216     static final Comparator<ChronoYearMonthDayHour> YEAR_MONTH_DAY_HOUR_ORDER =
217         (Comparator<ChronoYearMonthDayHour> & Serializable) (yearMonthDayHour1, yearMonthDayHour2) -> {
218             return Long.compare(yearMonthDayHour1.getYear(), yearMonthDayHour2.getYear());
219         };
220
221     /**
222      * ChronoYearMonthDayHourMinute order constant.
223      */
224     static final Comparator<ChronoYearMonthDayHourMinute> YEAR_MONTH_DAY_HOUR_MINUTE_ORDER =
225         (Comparator<ChronoYearMonthDayHourMinute> & Serializable) (yearMonthDayHourMinute1, yearMonthDayHourMinute2) -> {
226             return Long.compare(yearMonthDayHourMinute1.getYear(), yearMonthDayHourMinute2.getYear());
227         };
228
229     /**
230      * ChronoYearMonthDayHourMinuteSecond order constant.
231      */
232     static final Comparator<ChronoYearMonthDayHourMinuteSecond> YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_ORDER =
233         (Comparator<ChronoYearMonthDayHourMinuteSecond> & Serializable) (yearMonthDayHourMinuteSecond1, yearMonthDayHourMinuteSecond2) -> {
234             return Long.compare(yearMonthDayHourMinuteSecond1.getYear(), yearMonthDayHourMinuteSecond2.getYear());
235         };
236
237     /**
238      * ChronoYearMonthDayHourMinuteSecondNano order constant.
239      */
240     static final Comparator<ChronoYearMonthDayHourMinuteSecondNano> YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_ORDER =
241         (Comparator<ChronoYearMonthDayHourMinuteSecondNano> & Serializable) (yearMonthDayHourMinuteSecondNano1, yearMonthDayHourMinuteSecondNano2) -> {
242             return Long.compare(yearMonthDayHourMinuteSecondNano1.getYear(), yearMonthDayHourMinuteSecondNano2.getYear());
243         };
244
245     /**
246      * ChronoYearMonthDayHourMinuteSecondNanoOfTheDay order constant.
247      */
248     static final Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheDay> YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_OF_THE_DAY_ORDER =
249         (Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheDay> & Serializable) (yearMonthDayHourMinuteSecondNanoOfTheDay1, yearMonthDayHourMinuteSecondNanoOfTheDay2) -> {
250             return Long.compare(yearMonthDayHourMinuteSecondNanoOfTheDay1.getYear(), yearMonthDayHourMinuteSecondNanoOfTheDay2.getYear());
251         };
252
253     /**
254      * ChronoYearMonthDayHourMinuteSecondNanoOfTheMonth order constant.
255      */
256     static final Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheMonth> YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_OF_THE_MONTH_ORDER =
257         (Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheMonth> & Serializable) (yearMonthDayHourMinuteSecondNanoOfTheMonth1, yearMonthDayHourMinuteSecondNanoOfTheMonth2) -> {
258             return Long.compare(yearMonthDayHourMinuteSecondNanoOfTheMonth1.getYear(), yearMonthDayHourMinuteSecondNanoOfTheMonth2.getYear());
259         };
260
261     /**
262      * ChronoYearMonthDayHourMinuteSecondNanoOfTheYear order constant.
263      */
264     static final Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYear> YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_OF_THE_YEAR_ORDER =
265         (Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYear> & Serializable) (yearMonthDayHourMinuteSecondNanoOfTheYear1, yearMonthDayHourMinuteSecondNanoOfTheYear2) -> {
266             return Long.compare(yearMonthDayHourMinuteSecondNanoOfTheYear1.getYear(), yearMonthDayHourMinuteSecondNanoOfTheYear2.getYear());
267         };
268
269     /**
270      * ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonth order constant.
271      */
272     static final Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonth> YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_OF_THE_YEAR_MONTH_ORDER =
273         (Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonth> & Serializable) (yearMonthDayHourMinuteSecondNanoOfTheYearMonth1, yearMonthDayHourMinuteSecondNanoOfTheYearMonth2) -> {
274             return Long.compare(yearMonthDayHourMinuteSecondNanoOfTheYearMonth1.getYear(), yearMonthDayHourMinuteSecondNanoOfTheYearMonth2.getYear());
275         };
276
277     /**
278      * ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDay order constant.
279      */
280     static final Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDay> YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_OF_THE_YEAR_MONTH_DAY_ORDER =
281         (Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDay> & Serializable) (yearMonthDayHourMinuteSecondNanoOfTheYearMonthDay1, yearMonthDayHourMinuteSecondNanoOfTheYearMonthDay2) -> {
282             return Long.compare(yearMonthDayHourMinuteSecondNanoOfTheYearMonthDay1.getYear(), yearMonthDayHourMinuteSecondNanoOfTheYearMonthDay2.getYear());
283         };
284
285     /**
286      * ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHour order constant.
287      */
288     static final Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHour> YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_OF_THE_YEAR_MONTH_DAY_HOUR_ORDER =
289         (Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHour> & Serializable) (yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHour1, yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHour2) -> {
290             return Long.compare(yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHour1.getYear(), yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHour2.getYear());
291         };
292
293     /**
294      * ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinute order constant.
295      */
296     static final Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinute> YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_OF_THE_YEAR_MONTH_DAY_HOUR_MINUTE_ORDER =
297         (Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinute> & Serializable) (yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinute1, yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinute2) -> {
298             return Long.compare(yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinute1.getYear(), yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinute2.getYear());
299         };
300
301     /**
302      * ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecond order constant.
303      */
304     static final Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecond> YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_OF_THE_YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_ORDER =
305         (Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecond> & Serializable) (yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecond1, yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecond2) -> {
306             return Long.compare(yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecond1.getYear(), yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecond2.getYear());
307         };
308
309     /**
310      * ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNano order constant.
311      */
312     static final Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNano> YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_OF_THE_YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_ORDER =
313         (Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNano> & Serializable) (yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNano1, yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNano2) -> {
314             return Long.compare(yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNano1.getYear(), yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNano2.getYear());
315         };
316
317     /**
318      * ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNanoOfTheDay order constant.
319      */
320     static final Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNanoOfTheDay> YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_OF_THE_YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_OF_THE_DAY_ORDER =
321         (Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNanoOfTheDay> & Serializable) (yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNanoOfTheDay1, yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNanoOfTheDay2) -> {
322             return Long.compare(yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNanoOfTheDay1.getYear(), yearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNanoOfTheDay2.getYear());
323         };
324
325     /**
326      * ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNanoOfTheMonth order constant.
327      */
328     static final Comparator<ChronoYearMonthDayHourMinuteSecondNanoOfTheYearMonthDayHourMinuteSecondNanoOfTheMonth> YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_OF_THE_YEAR_MONTH_DAY_HOUR_MINUTE_SECOND_NANO_OF_THE_MONTH_ORDER =
329         (Comparator<ChronoYear
```

```

155         }
156         return cmp;
157     };
158
159     /**
160      * Map of available calendars by ID.
161      */
162     private static final ConcurrentHashMap<String, Chronology> CHRONOS_BY_ID = new
        ConcurrentHashMap<>();
163
164     /**
165      * Map of available calendars by calendar type.
166      */
167     private static final ConcurrentHashMap<String, Chronology> CHRONOS_BY_TYPE = new
        ConcurrentHashMap<>();
168
169     /**
170      * Register a Chronology by its ID and type for lookup by {@link #of(String)}.
171      * Chronologies must not be registered until they are completely constructed.
172      * Specifically, not in the constructor of Chronology.
173      *
174      * @param chrono the chronology to register; not null
175      * @return the already registered Chronology if any, may be null
176      */
177     static Chronology registerChrono(Chronology chrono) {
178         return registerChrono(chrono, chrono.getId());
179     }
180
181     /**
182      * Register a Chronology by ID and type for lookup by {@link #of(String)}.
183      * Chronos must not be registered until they are completely constructed.
184      * Specifically, not in the constructor of Chronology.
185      *
186      * @param chrono the chronology to register; not null
187      * @param id the ID to register the chronology; not null
188      * @return the already registered Chronology if any, may be null
189      */
190     static Chronology registerChrono(Chronology chrono, String id) {
191         Chronology prev = CHRONOS_BY_ID.putIfAbsent(id, chrono);
192         if (prev == null) {
193             String type = chrono.getCalendarType();
194             if (type != null) {
195                 CHRONOS_BY_TYPE.putIfAbsent(type, chrono);
196             }
197         }
198         return prev;
199     }
200
201     /**
202      * Initialization of the maps from id and type to Chronology.
203      * The ServiceLoader is used to find and register any implementations
204      * of {@link java.time.chrono.AbstractChronology} found in the bootclass loader.
205      * The built-in chronologies are registered explicitly.
206      * Calendars configured via the Thread's context classloader are local

```



```

206     * to that thread and are ignored.
207     * <p>
208     * The initialization is done only once using the registration
209     * of the IsoChronology as the test and the final step.
210     * Multiple threads may perform the initialization concurrently.
211     * Only the first registration of each Chronology is retained by the
212     * ConcurrentHashMap.
213     * @return true if the cache was initialized
214     */
215     private static boolean initCache() {
216         if (CHRONOS_BY_ID.get("ISO") == null) {
217             // Initialization is incomplete
218
219             // Register built-in Chronologies
220             registerChrono(HijrahChronology.INSTANCE);
221             registerChrono(JapaneseChronology.INSTANCE);
222             registerChrono(MinguoChronology.INSTANCE);
223             registerChrono(ThaiBuddhistChronology.INSTANCE);
224
225             // Register Chronologies from the ServiceLoader
226             @SuppressWarnings("rawtypes")
227             ServiceLoader<AbstractChronology> loader = ServiceLoader.load(AbstractChronology.class
228                                     , null);
229             for (AbstractChronology chrono : loader) {
230                 String id = chrono.getId();
231                 if (id.equals("ISO") || registerChrono(chrono) != null) {
232                     // Log the attempt to replace an existing Chronology
233                     PlatformLogger logger = PlatformLogger.getLogger("java.time.chrono");
234                     logger.warning("Ignoring duplicate Chronology, from ServiceLoader configuration
235                                   " + id);
236                 }
237             }
238
239             // finally, register IsoChronology to mark initialization is complete
240             registerChrono(IsoChronology.INSTANCE);
241             return true;
242         }
243         return false;
244     }
245
246     /**
247     * Obtains an instance of {@code Chronology} from a locale.
248     * <p>
249     * See {@link Chronology#of(Locale)}.
250     *
251     * @param locale the locale to use to obtain the calendar system, not null
252     * @return the calendar system associated with the locale, not null
253     * @throws java.time.DateTimeException if the locale-specified calendar cannot be found
254     */
255     static Chronology of(Locale locale) {
256         Objects.requireNonNull(locale, "locale");
257         String type = locale.getUnicodeLocaleType("ca");

```

```

257     if (type == null || "iso".equals(type) || "iso8601".equals(type)) {
258         return IsoChronology.INSTANCE;
259     }
260     // Not pre-defined; lookup by the type
261     do {
262         Chronology chrono = CHRONOS_BY_TYPE.get(type);
263         if (chrono != null) {
264             return chrono;
265         }
266         // If not found, do the initialization (once) and repeat the lookup
267     } while (initCache());
268
269     // Look for a Chronology using ServiceLoader of the Thread's ContextClassLoader
270     // Application provided Chronologies must not be cached
271     @SuppressWarnings("rawtypes")
272     ServiceLoader<Chronology> loader = ServiceLoader.load(Chronology.class);
273     for (Chronology chrono : loader) {
274         if (type.equals(chrono.getCalendarType())) {
275             return chrono;
276         }
277     }
278     throw new DateTimeException("Unknown calendar system: " + type);
279 }
280
281 //-----
282 /**
283  * Obtains an instance of {@code Chronology} from a chronology ID or
284  * calendar system type.
285  * <p>
286  * See {@link Chronology#of(String)}.
287  *
288  * @param id the chronology ID or calendar system type, not null
289  * @return the chronology with the identifier requested, not null
290  * @throws java.time.DateTimeException if the chronology cannot be found
291  */
292 static Chronology of(String id) {
293     Objects.requireNonNull(id, "id");
294     do {
295         Chronology chrono = of0(id);
296         if (chrono != null) {
297             return chrono;
298         }
299         // If not found, do the initialization (once) and repeat the lookup
300     } while (initCache());
301
302     // Look for a Chronology using ServiceLoader of the Thread's ContextClassLoader
303     // Application provided Chronologies must not be cached
304     @SuppressWarnings("rawtypes")
305     ServiceLoader<Chronology> loader = ServiceLoader.load(Chronology.class);
306     for (Chronology chrono : loader) {
307         if (id.equals(chrono.getId()) || id.equals(chrono.getCalendarType())) {
308             return chrono;
309         }

```

```

310     }
311     throw new DateTimeException("Unknown chronology: " + id);
312 }
313
314 /**
315  * Obtains an instance of {@code Chronology} from a chronology ID or
316  * calendar system type.
317  *
318  * @param id the chronology ID or calendar system type, not null
319  * @return the chronology with the identifier requested, or {@code null} if not found
320  */
321 private static Chronology of0(String id) {
322     Chronology chrono = CHRONOS_BY_ID.get(id);
323     if (chrono == null) {
324         chrono = CHRONOS_BY_TYPE.get(id);
325     }
326     return chrono;
327 }
328
329 /**
330  * Returns the available chronologies.
331  * <p>
332  * Each returned {@code Chronology} is available for use in the system.
333  * The set of chronologies includes the system chronologies and
334  * any chronologies provided by the application via ServiceLoader
335  * configuration.
336  *
337  * @return the independent, modifiable set of the available chronology IDs, not null
338  */
339 static Set<Chronology> getAvailableChronologies() {
340     initCache(); // force initialization
341     HashSet<Chronology> chronos = new HashSet<>(CHRONOS_BY_ID.values());
342
343     /// Add in Chronologies from the ServiceLoader configuration
344     @SuppressWarnings("rawtypes")
345     ServiceLoader<Chronology> loader = ServiceLoader.load(Chronology.class);
346     for (Chronology chrono : loader) {
347         chronos.add(chrono);
348     }
349     return chronos;
350 }
351
352 //-----
353 /**
354  * Creates an instance.
355  */
356 protected AbstractChronology() {
357 }
358
359 //-----
360 /**
361  * Resolves parsed {@code ChronoField} values into a date during parsing.
362  * <p>

```

```

363 * Most {@code TemporalField} implementations are resolved using the
364 * resolve method on the field. By contrast, the {@code ChronoField} class
365 * defines fields that only have meaning relative to the chronology.
366 * As such, {@code ChronoField} date fields are resolved here in the
367 * context of a specific chronology.
368 * <p>
369 * {@code ChronoField} instances are resolved by this method, which may
370 * be overridden in subclasses.
371 * <ul>
372 * <li>{@code EPOCH_DAY} - If present, this is converted to a date and
373 * all other date fields are then cross-checked against the date.
374 * <li>{@code PROLEPTIC_MONTH} - If present, then it is split into the
375 * {@code YEAR} and {@code MONTH_OF_YEAR}. If the mode is strict or smart
376 * then the field is validated.
377 * <li>{@code YEAR_OF_ERA} and {@code ERA} - If both are present, then they
378 * are combined to form a {@code YEAR}. In lenient mode, the {@code YEAR_OF_ERA}
379 * range is not validated, in smart and strict mode it is. The {@code ERA} is
380 * validated for range in all three modes. If only the {@code YEAR_OF_ERA} is
381 * present, and the mode is smart or lenient, then the last available era
382 * is assumed. In strict mode, no era is assumed and the {@code YEAR_OF_ERA} is
383 * left untouched. If only the {@code ERA} is present, then it is left untouched.
384 * <li>{@code YEAR}, {@code MONTH_OF_YEAR} and {@code DAY_OF_MONTH} -
385 * If all three are present, then they are combined to form a date.
386 * In all three modes, the {@code YEAR} is validated.
387 * If the mode is smart or strict, then the month and day are validated.
388 * If the mode is lenient, then the date is combined in a manner equivalent to
389 * creating a date on the first day of the first month in the requested year,
390 * then adding the difference in months, then the difference in days.
391 * If the mode is smart, and the day-of-month is greater than the maximum for
392 * the year-month, then the day-of-month is adjusted to the last day-of-month.
393 * If the mode is strict, then the three fields must form a valid date.
394 * <li>{@code YEAR} and {@code DAY_OF_YEAR} -
395 * If both are present, then they are combined to form a date.
396 * In all three modes, the {@code YEAR} is validated.
397 * If the mode is lenient, then the date is combined in a manner equivalent to
398 * creating a date on the first day of the requested year, then adding
399 * the difference in days.
400 * If the mode is smart or strict, then the two fields must form a valid date.
401 * <li>{@code YEAR}, {@code MONTH_OF_YEAR}, {@code ALIGNED_WEEK_OF_MONTH} and
402 * {@code ALIGNED_DAY_OF_WEEK_IN_MONTH} -
403 * If all four are present, then they are combined to form a date.
404 * In all three modes, the {@code YEAR} is validated.
405 * If the mode is lenient, then the date is combined in a manner equivalent to
406 * creating a date on the first day of the first month in the requested year, then adding
407 * the difference in months, then the difference in weeks, then in days.
408 * If the mode is smart or strict, then the all four fields are validated to
409 * their outer ranges. The date is then combined in a manner equivalent to
410 * creating a date on the first day of the requested year and month, then adding
411 * the amount in weeks and days to reach their values. If the mode is strict,
412 * the date is additionally validated to check that the day and week adjustment
413 * did not change the month.
414 * <li>{@code YEAR}, {@code MONTH_OF_YEAR}, {@code ALIGNED_WEEK_OF_MONTH} and
415 * {@code DAY_OF_WEEK} - If all four are present, then they are combined to

```

```

416 * form a date. The approach is the same as described above for
417 * years, months and weeks in {@code ALIGNED_DAY_OF_WEEK_IN_MONTH}.
418 * The day-of-week is adjusted as the next or same matching day-of-week once
419 * the years, months and weeks have been handled.
420 * <li>{@code YEAR}, {@code ALIGNED_WEEK_OF_YEAR} and {@code ALIGNED_DAY_OF_WEEK_IN_YEAR} -
421 * If all three are present, then they are combined to form a date.
422 * In all three modes, the {@code YEAR} is validated.
423 * If the mode is lenient, then the date is combined in a manner equivalent to
424 * creating a date on the first day of the requested year, then adding
425 * the difference in weeks, then in days.
426 * If the mode is smart or strict, then the all three fields are validated to
427 * their outer ranges. The date is then combined in a manner equivalent to
428 * creating a date on the first day of the requested year, then adding
429 * the amount in weeks and days to reach their values. If the mode is strict,
430 * the date is additionally validated to check that the day and week adjustment
431 * did not change the year.
432 * <li>{@code YEAR}, {@code ALIGNED_WEEK_OF_YEAR} and {@code DAY_OF_WEEK} -
433 * If all three are present, then they are combined to form a date.
434 * The approach is the same as described above for years and weeks in
435 * {@code ALIGNED_DAY_OF_WEEK_IN_YEAR}. The day-of-week is adjusted as the
436 * next or same matching day-of-week once the years and weeks have been handled.
437 * </ul>
438 * <p>
439 * The default implementation is suitable for most calendar systems.
440 * If {@link java.time.temporal.ChronoField#YEAR_OF_ERA} is found without an {@link java.time.
    temporal.ChronoField#ERA}
441 * then the last era in {@link #eras()} is used.
442 * The implementation assumes a 7 day week, that the first day-of-month
443 * has the value 1, that first day-of-year has the value 1, and that the
444 * first of the month and year always exists.
445 *
446 * @param fieldValues the map of fields to values, which can be updated, not null
447 * @param resolverStyle the requested type of resolve, not null
448 * @return the resolved date, null if insufficient information to create a date
449 * @throws java.time.DateTimeException if the date cannot be resolved, typically
450 * because of a conflict in the input data
451 */
452 @Override
453 public ChronoLocalDate resolveDate(Map<TemporalField, Long> fieldValues, ResolverStyle
    resolverStyle) {
454     // check epoch-day before inventing era
455     if (fieldValues.containsKey(EPOCH_DAY)) {
456         return dateEpochDay(fieldValues.remove(EPOCH_DAY));
457     }
458
459     // fix proleptic month before inventing era
460     resolveProlepticMonth(fieldValues, resolverStyle);
461
462     // invent era if necessary to resolve year-of-era
463     ChronoLocalDate resolved = resolveYearOfEra(fieldValues, resolverStyle);
464     if (resolved != null) {
465         return resolved;
466     }

```

```

467
468 // build date
469 if (fieldValues.containsKey(YEAR)) {
470     if (fieldValues.containsKey(MONTH_OF_YEAR)) {
471         if (fieldValues.containsKey(DAY_OF_MONTH)) {
472             return resolveYMD(fieldValues, resolverStyle);
473         }
474         if (fieldValues.containsKey(ALIGNED_WEEK_OF_MONTH)) {
475             if (fieldValues.containsKey(ALIGNED_DAY_OF_WEEK_IN_MONTH)) {
476                 return resolveYMAA(fieldValues, resolverStyle);
477             }
478             if (fieldValues.containsKey(DAY_OF_WEEK)) {
479                 return resolveYMAD(fieldValues, resolverStyle);
480             }
481         }
482     }
483     if (fieldValues.containsKey(DAY_OF_YEAR)) {
484         return resolveYD(fieldValues, resolverStyle);
485     }
486     if (fieldValues.containsKey(ALIGNED_WEEK_OF_YEAR)) {
487         if (fieldValues.containsKey(ALIGNED_DAY_OF_WEEK_IN_YEAR)) {
488             return resolveYAA(fieldValues, resolverStyle);
489         }
490         if (fieldValues.containsKey(DAY_OF_WEEK)) {
491             return resolveYAD(fieldValues, resolverStyle);
492         }
493     }
494 }
495 return null;
496 }
497
498 void resolveProlepticMonth(Map<TemporalField, Long> fieldValues, ResolverStyle resolverStyle) {
499     Long pMonth = fieldValues.remove(PROLEPTIC_MONTH);
500     if (pMonth != null) {
501         if (resolverStyle != ResolverStyle.LENIENT) {
502             PROLEPTIC_MONTH.checkValidValue(pMonth);
503         }
504         // first day-of-month is likely to be safest for setting proleptic-month
505         // cannot add to year zero, as not all chronologies have a year zero
506         ChronoLocalDate chronoDate = dateNow()
507             .with(DAY_OF_MONTH, 1).with(PROLEPTIC_MONTH, pMonth);
508         addFieldValue(fieldValues, MONTH_OF_YEAR, chronoDate.get(MONTH_OF_YEAR));
509         addFieldValue(fieldValues, YEAR, chronoDate.get(YEAR));
510     }
511 }
512
513 ChronoLocalDate resolveYearOfEra(Map<TemporalField, Long> fieldValues, ResolverStyle
514     resolverStyle) {
515     Long yoeLong = fieldValues.remove(YEAR_OF_ERA);
516     if (yoeLong != null) {
517         Long eraLong = fieldValues.remove(ERA);
518         int yoe;
519         if (resolverStyle != ResolverStyle.LENIENT) {

```

```

519         yoe = range(YEAR_OF_ERA).checkValidIntValue(yoeLong, YEAR_OF_ERA);
520     } else {
521         yoe = Math.toIntExact(yoeLong);
522     }
523     if (eraLong != null) {
524         Era eraObj = eraOf(range(ERA).checkValidIntValue(eraLong, ERA));
525         addFieldValue(fieldValues, YEAR, prolepticYear(eraObj, yoe));
526     } else {
527         if (fieldValues.containsKey(YEAR)) {
528             int year = range(YEAR).checkValidIntValue(fieldValues.get(YEAR), YEAR);
529             ChronoLocalDate chronoDate = dateYearDay(year, 1);
530             addFieldValue(fieldValues, YEAR, prolepticYear(chronoDate.getEra(), yoe));
531         } else if (resolverStyle == ResolverStyle.STRICT) {
532             // do not invent era if strict
533             // reinstate the field removed earlier, no cross-check issues
534             fieldValues.put(YEAR_OF_ERA, yoeLong);
535         } else {
536             List<Era> eras = eras();
537             if (eras.isEmpty()) {
538                 addFieldValue(fieldValues, YEAR, yoe);
539             } else {
540                 Era eraObj = eras.get(eras.size() - 1);
541                 addFieldValue(fieldValues, YEAR, prolepticYear(eraObj, yoe));
542             }
543         }
544     }
545     } else if (fieldValues.containsKey(ERA)) {
546         range(ERA).checkValidValue(fieldValues.get(ERA), ERA); // always validated
547     }
548     return null;
549 }
550
551 ChronoLocalDate resolveYMD(Map<TemporalField, Long> fieldValues, ResolverStyle resolverStyle) {
552     int y = range(YEAR).checkValidIntValue(fieldValues.remove(YEAR), YEAR);
553     if (resolverStyle == ResolverStyle.LENIENT) {
554         long months = Math.subtractExact(fieldValues.remove(MONTH_OF_YEAR), 1);
555         long days = Math.subtractExact(fieldValues.remove(DAY_OF_MONTH), 1);
556         return date(y, 1, 1).plus(months, MONTHS).plus(days, DAYS);
557     }
558     int moy = range(MONTH_OF_YEAR).checkValidIntValue(fieldValues.remove(MONTH_OF_YEAR),
559         MONTH_OF_YEAR);
559     ValueRange domRange = range(DAY_OF_MONTH);
560     int dom = domRange.checkValidIntValue(fieldValues.remove(DAY_OF_MONTH), DAY_OF_MONTH);
561     if (resolverStyle == ResolverStyle.SMART) { // previous valid
562         try {
563             return date(y, moy, dom);
564         } catch (DateTimeException ex) {
565             return date(y, moy, 1).with(TemporalAdjusters.lastDayOfMonth());
566         }
567     }
568     return date(y, moy, dom);
569 }
570

```



```

571 ChronoLocalDate resolveYD(Map<TemporalField, Long> fieldValues, ResolverStyle resolverStyle) {
572     int y = range(YEAR).checkValidIntValue(fieldValues.remove(YEAR), YEAR);
573     if (resolverStyle == ResolverStyle.LENIENT) {
574         long days = Math.subtractExact(fieldValues.remove(DAY_OF_YEAR), 1);
575         return dateYearDay(y, 1).plus(days, DAYS);
576     }
577     int doy = range(DAY_OF_YEAR).checkValidIntValue(fieldValues.remove(DAY_OF_YEAR),
578         DAY_OF_YEAR);
579     return dateYearDay(y, doy); // smart is same as strict
580 }
581 ChronoLocalDate resolveYMAA(Map<TemporalField, Long> fieldValues, ResolverStyle resolverStyle)
582 {
583     int y = range(YEAR).checkValidIntValue(fieldValues.remove(YEAR), YEAR);
584     if (resolverStyle == ResolverStyle.LENIENT) {
585         long months = Math.subtractExact(fieldValues.remove(MONTH_OF_YEAR), 1);
586         long weeks = Math.subtractExact(fieldValues.remove(ALIGNED_WEEK_OF_MONTH), 1);
587         long days = Math.subtractExact(fieldValues.remove(ALIGNED_DAY_OF_WEEK_IN_MONTH), 1);
588         return date(y, 1, 1).plus(months, MONTHS).plus(weeks, WEEKS).plus(days, DAYS);
589     }
590     int moy = range(MONTH_OF_YEAR).checkValidIntValue(fieldValues.remove(MONTH_OF_YEAR),
591         MONTH_OF_YEAR);
592     int aw = range(ALIGNED_WEEK_OF_MONTH).checkValidIntValue(fieldValues.remove(
593         ALIGNED_WEEK_OF_MONTH), ALIGNED_WEEK_OF_MONTH);
594     int ad = range(ALIGNED_DAY_OF_WEEK_IN_MONTH).checkValidIntValue(fieldValues.remove(
595         ALIGNED_DAY_OF_WEEK_IN_MONTH), ALIGNED_DAY_OF_WEEK_IN_MONTH);
596     ChronoLocalDate date = date(y, moy, 1).plus((aw - 1) * 7 + (ad - 1), DAYS);
597     if (resolverStyle == ResolverStyle.STRICT && date.get(MONTH_OF_YEAR) != moy) {
598         throw new DateTimeException("Strict mode rejected resolved date as it is in a different
599             month");
600     }
601     return date;
602 }
603 ChronoLocalDate resolveYMAD(Map<TemporalField, Long> fieldValues, ResolverStyle resolverStyle)
604 {
605     int y = range(YEAR).checkValidIntValue(fieldValues.remove(YEAR), YEAR);
606     if (resolverStyle == ResolverStyle.LENIENT) {
607         long months = Math.subtractExact(fieldValues.remove(MONTH_OF_YEAR), 1);
608         long weeks = Math.subtractExact(fieldValues.remove(ALIGNED_WEEK_OF_MONTH), 1);
609         long dow = Math.subtractExact(fieldValues.remove(DAY_OF_WEEK), 1);
610         return resolveAligned(date(y, 1, 1), months, weeks, dow);
611     }
612     int moy = range(MONTH_OF_YEAR).checkValidIntValue(fieldValues.remove(MONTH_OF_YEAR),
613         MONTH_OF_YEAR);
614     int aw = range(ALIGNED_WEEK_OF_MONTH).checkValidIntValue(fieldValues.remove(
615         ALIGNED_WEEK_OF_MONTH), ALIGNED_WEEK_OF_MONTH);
616     int dow = range(DAY_OF_WEEK).checkValidIntValue(fieldValues.remove(DAY_OF_WEEK),
617         DAY_OF_WEEK);
618     ChronoLocalDate date = date(y, moy, 1).plus((aw - 1) * 7, DAYS).with(nextOrSame(DayOfWeek.
619         of(dow)));
620     if (resolverStyle == ResolverStyle.STRICT && date.get(MONTH_OF_YEAR) != moy) {
621         throw new DateTimeException("Strict mode rejected resolved date as it is in a different

```



```

        month");
613     }
614     return date;
615 }
616
617 ChronoLocalDate resolveYAA(Map<TemporalField, Long> fieldValues, ResolverStyle resolverStyle) {
618     int y = range(YEAR).checkValidIntValue(fieldValues.remove(YEAR), YEAR);
619     if (resolverStyle == ResolverStyle.LENIENT) {
620         long weeks = Math.subtractExact(fieldValues.remove(ALIGNED_WEEK_OF_YEAR), 1);
621         long days = Math.subtractExact(fieldValues.remove(ALIGNED_DAY_OF_WEEK_IN_YEAR), 1);
622         return dateYearDay(y, 1).plus(weeks, WEEKS).plus(days, DAYS);
623     }
624     int aw = range(ALIGNED_WEEK_OF_YEAR).checkValidIntValue(fieldValues.remove(
        ALIGNED_WEEK_OF_YEAR), ALIGNED_WEEK_OF_YEAR);
625     int ad = range(ALIGNED_DAY_OF_WEEK_IN_YEAR).checkValidIntValue(fieldValues.remove(
        ALIGNED_DAY_OF_WEEK_IN_YEAR), ALIGNED_DAY_OF_WEEK_IN_YEAR);
626     ChronoLocalDate date = dateYearDay(y, 1).plus((aw - 1) * 7 + (ad - 1), DAYS);
627     if (resolverStyle == ResolverStyle.STRICT && date.get(YEAR) != y) {
628         throw new DateTimeException("Strict mode rejected resolved date as it is in a different
        year");
629     }
630     return date;
631 }
632
633 ChronoLocalDate resolveYAD(Map<TemporalField, Long> fieldValues, ResolverStyle resolverStyle) {
634     int y = range(YEAR).checkValidIntValue(fieldValues.remove(YEAR), YEAR);
635     if (resolverStyle == ResolverStyle.LENIENT) {
636         long weeks = Math.subtractExact(fieldValues.remove(ALIGNED_WEEK_OF_YEAR), 1);
637         long dow = Math.subtractExact(fieldValues.remove(DAY_OF_WEEK), 1);
638         return resolveAligned(dateYearDay(y, 1), 0, weeks, dow);
639     }
640     int aw = range(ALIGNED_WEEK_OF_YEAR).checkValidIntValue(fieldValues.remove(
        ALIGNED_WEEK_OF_YEAR), ALIGNED_WEEK_OF_YEAR);
641     int dow = range(DAY_OF_WEEK).checkValidIntValue(fieldValues.remove(DAY_OF_WEEK),
        DAY_OF_WEEK);
642     ChronoLocalDate date = dateYearDay(y, 1).plus((aw - 1) * 7, DAYS).with(nextOrSame(DayOfWeek
        .of(dow)));
643     if (resolverStyle == ResolverStyle.STRICT && date.get(YEAR) != y) {
644         throw new DateTimeException("Strict mode rejected resolved date as it is in a different
        year");
645     }
646     return date;
647 }
648
649 ChronoLocalDate resolveAligned(ChronoLocalDate base, long months, long weeks, long dow) {
650     ChronoLocalDate date = base.plus(months, MONTHS).plus(weeks, WEEKS);
651     if (dow > 7) {
652         date = date.plus((dow - 1) / 7, WEEKS);
653         dow = ((dow - 1) % 7) + 1;
654     } else if (dow < 1) {
655         date = date.plus(Math.subtractExact(dow, 7) / 7, WEEKS);
656         dow = ((dow + 6) % 7) + 1;
657     }

```

```

658         return date.with(nextOrSame(DayOfWeek.of((int) dow)));
659     }
660
661     /**
662      * Adds a field-value pair to the map, checking for conflicts.
663      * <p>
664      * If the field is not already present, then the field-value pair is added to the map.
665      * If the field is already present and it has the same value as that specified, no action
666      * occurs.
667      * If the field is already present and it has a different value to that specified, then
668      * an exception is thrown.
669      *
670      * @param field the field to add, not null
671      * @param value the value to add, not null
672      * @throws java.time.DateTimeException if the field is already present with a different value
673      */
674     void addFieldValue(Map<TemporalField, Long> fieldValues, ChronoField field, long value) {
675         Long old = fieldValues.get(field); // check first for better error message
676         if (old != null && old.longValue() != value) {
677             throw new DateTimeException("Conflict found: " + field + " " + old + " differs from " +
678                 field + " " + value);
679         }
680         fieldValues.put(field, value);
681     }
682
683     //-----
684     /**
685      * Compares this chronology to another chronology.
686      * <p>
687      * The comparison order first by the chronology ID string, then by any
688      * additional information specific to the subclass.
689      * It is "consistent with equals", as defined by {@link Comparable}.
690      *
691      * @implSpec
692      * This implementation compares the chronology ID.
693      * Subclasses must compare any additional state that they store.
694      *
695      * @param other the other chronology to compare to, not null
696      * @return the comparator value, negative if less, positive if greater
697      */
698     @Override
699     public int compareTo(Chronology other) {
700         return getId().compareTo(other.getId());
701     }
702
703     /**
704      * Checks if this chronology is equal to another chronology.
705      * <p>
706      * The comparison is based on the entire state of the object.
707      *
708      * @implSpec
709      * This implementation checks the type and calls
710      * {@link #compareTo(java.time.chrono.Chronology)}.

```

```

709      *
710      * @param obj the object to check, null returns false
711      * @return true if this is equal to the other chronology
712      */
713      @Override
714      public boolean equals(Object obj) {
715          if (this == obj) {
716              return true;
717          }
718          if (obj instanceof AbstractChronology) {
719              return compareTo((AbstractChronology) obj) == 0;
720          }
721          return false;
722      }
723
724      /**
725       * A hash code for this chronology.
726       * <p>
727       * The hash code should be based on the entire state of the object.
728       *
729       * @implSpec
730       * This implementation is based on the chronology ID and class.
731       * Subclasses should add any additional state that they store.
732       *
733       * @return a suitable hash code
734       */
735      @Override
736      public int hashCode() {
737          return getClass().hashCode() ^ getId().hashCode();
738      }
739
740      //-----
741      /**
742       * Outputs this chronology as a {@code String}, using the chronology ID.
743       *
744       * @return a string representation of this chronology, not null
745       */
746      @Override
747      public String toString() {
748          return getId();
749      }
750
751      //-----
752      /**
753       * Writes the Chronology using a
754       * <a href="../../serialized-form.html#java.time.chrono.Ser">dedicated serialized form</a>.
755       * <pre>
756       * out.writeByte(1); // identifies this as a Chronology
757       * out.writeUTF(getId());
758       * </pre>
759       *
760       * @return the instance of {@code Ser}, not null
761       */

```

```

762     Object writeReplace() {
763         return new Ser(Ser.CHRONO_TYPE, this);
764     }
765
766     /**
767      * Defend against malicious streams.
768      *
769      * @param s the stream to read
770      * @throws java.io.InvalidObjectException always
771      */
772     private void readObject(ObjectInputStream s) throws ObjectStreamException {
773         throw new InvalidObjectException("Deserialization via serialization delegate");
774     }
775
776     void writeExternal(DataOutput out) throws IOException {
777         out.writeUTF(getId());
778     }
779
780     static Chronology readExternal(DataInput in) throws IOException {
781         String id = in.readUTF();
782         return Chronology.of(id);
783     }
784
785 }

```

5.2 ChronoLocalDate.java

Secuencia 72: java/time/chrono/ChronoLocalDate.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25

```

```
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.chrono;
63
64  import static java.time.temporal.ChronoField.EPOCH_DAY;
65  import static java.time.temporal.ChronoField.ERA;
66  import static java.time.temporal.ChronoField.YEAR;
67  import static java.time.temporal.ChronoUnit.DAYS;
68
69  import java.time.DateTimeException;
70  import java.time.LocalDate;
71  import java.time.LocalDateTime;
72  import java.time.format.DateTimeFormatter;
73  import java.time.temporal.ChronoField;
74  import java.time.temporal.ChronoUnit;
75  import java.time.temporal.Temporal;
76  import java.time.temporal.TemporalAccessor;
77  import java.time.temporal.TemporalAdjuster;
78  import java.time.temporal.TemporalAmount;
```

```

79 import java.time.temporal.TemporalField;
80 import java.time.temporal.TemporalQueries;
81 import java.time.temporal.TemporalQuery;
82 import java.time.temporal.TemporalUnit;
83 import java.time.temporal.UnsupportedTemporalTypeException;
84 import java.util.Comparator;
85 import java.util.Objects;
86
87 /**
88  * A date without time-of-day or time-zone in an arbitrary chronology, intended
89  * for advanced globalization use cases.
90  * <p>
91  * <b>Most applications should declare method signatures, fields and variables
92  * as {@link LocalDate}, not this interface.</b>
93  * <p>
94  * A {@code ChronoLocalDate} is the abstract representation of a date where the
95  * {@code Chronology chronology}, or calendar system, is pluggable.
96  * The date is defined in terms of fields expressed by {@link TemporalField},
97  * where most common implementations are defined in {@link ChronoField}.
98  * The chronology defines how the calendar system operates and the meaning of
99  * the standard fields.
100  *
101  * <h3>When to use this interface</h3>
102  * The design of the API encourages the use of {@code LocalDate} rather than this
103  * interface, even in the case where the application needs to deal with multiple
104  * calendar systems.
105  * <p>
106  * This concept can seem surprising at first, as the natural way to globalize an
107  * application might initially appear to be to abstract the calendar system.
108  * However, as explored below, abstracting the calendar system is usually the wrong
109  * approach, resulting in logic errors and hard to find bugs.
110  * As such, it should be considered an application-wide architectural decision to choose
111  * to use this interface as opposed to {@code LocalDate}.
112  *
113  * <h3>Architectural issues to consider</h3>
114  * These are some of the points that must be considered before using this interface
115  * throughout an application.
116  * <p>
117  * 1) Applications using this interface, as opposed to using just {@code LocalDate},
118  * face a significantly higher probability of bugs. This is because the calendar system
119  * in use is not known at development time. A key cause of bugs is where the developer
120  * applies assumptions from their day-to-day knowledge of the ISO calendar system
121  * to code that is intended to deal with any arbitrary calendar system.
122  * The section below outlines how those assumptions can cause problems
123  * The primary mechanism for reducing this increased risk of bugs is a strong code review process.
124  * This should also be considered a extra cost in maintenance for the lifetime of the code.
125  * <p>
126  * 2) This interface does not enforce immutability of implementations.
127  * While the implementation notes indicate that all implementations must be immutable
128  * there is nothing in the code or type system to enforce this. Any method declared
129  * to accept a {@code ChronoLocalDate} could therefore be passed a poorly or
130  * maliciously written mutable implementation.
131  * <p>

```

```

132 * 3) Applications using this interface must consider the impact of eras.
133 * {@code LocalDate} shields users from the concept of eras, by ensuring that {@code getYear()}
134 * returns the proleptic year. That decision ensures that developers can think of
135 * {@code LocalDate} instances as consisting of three fields - year, month-of-year and day-of-month
136 *
137 * By contrast, users of this interface must think of dates as consisting of four fields -
138 * era, year-of-era, month-of-year and day-of-month. The extra era field is frequently
139 * forgotten, yet it is of vital importance to dates in an arbitrary calendar system.
140 * For example, in the Japanese calendar system, the era represents the reign of an Emperor.
141 * Whenever one reign ends and another starts, the year-of-era is reset to one.
142 *
143 * <p>
144 * 4) The only agreed international standard for passing a date between two systems
145 * is the ISO-8601 standard which requires the ISO calendar system. Using this interface
146 * throughout the application will inevitably lead to the requirement to pass the date
147 * across a network or component boundary, requiring an application specific protocol or format.
148 *
149 * <p>
150 * 5) Long term persistence, such as a database, will almost always only accept dates in the
151 * ISO-8601 calendar system (or the related Julian-Gregorian). Passing around dates in other
152 * calendar systems increases the complications of interacting with persistence.
153 *
154 * <p>
155 * 6) Most of the time, passing a {@code ChronoLocalDate} throughout an application
156 * is unnecessary, as discussed in the last section below.
157 *
158 * <h3>False assumptions causing bugs in multi-calendar system code</h3>
159 * As indicated above, there are many issues to consider when try to use and manipulate a
160 * date in an arbitrary calendar system. These are some of the key issues.
161 *
162 * <p>
163 * Code that queries the day-of-month and assumes that the value will never be more than
164 * 31 is invalid. Some calendar systems have more than 31 days in some months.
165 *
166 * <p>
167 * Code that adds 12 months to a date and assumes that a year has been added is invalid.
168 * Some calendar systems have a different number of months, such as 13 in the Coptic or Ethiopic.
169 *
170 * <p>
171 * Code that adds one month to a date and assumes that the month-of-year value will increase
172 * by one or wrap to the next year is invalid. Some calendar systems have a variable number
173 * of months in a year, such as the Hebrew.
174 *
175 * <p>
176 * Code that adds one month, then adds a second one month and assumes that the day-of-month
177 * will remain close to its original value is invalid. Some calendar systems have a large
178 * difference
179 * between the length of the longest month and the length of the shortest month.
180 * For example, the Coptic or Ethiopic have 12 months of 30 days and 1 month of 5 days.
181 *
182 * <p>
183 * Code that adds seven days and assumes that a week has been added is invalid.
184 * Some calendar systems have weeks of other than seven days, such as the French Revolutionary.
185 *
186 * <p>
187 * Code that assumes that because the year of {@code date1} is greater than the year of {@code
188 * date2}
189 * then {@code date1} is after {@code date2} is invalid. This is invalid for all calendar systems
190 * when referring to the year-of-era, and especially untrue of the Japanese calendar system
191 * where the year-of-era restarts with the reign of every new Emperor.
192 *
193 * <p>
194 * Code that treats month-of-year one and day-of-month one as the start of the year is invalid.

```

```

182 * Not all calendar systems start the year when the month value is one.
183 * <p>
184 * In general, manipulating a date, and even querying a date, is wide open to bugs when the
185 * calendar system is unknown at development time. This is why it is essential that code using
186 * this interface is subjected to additional code reviews. It is also why an architectural
187 * decision to avoid this interface type is usually the correct one.
188 *
189 * <h3>Using LocalDate instead</h3>
190 * The primary alternative to using this interface throughout your application is as follows.
191 * <ul>
192 * <li>Declare all method signatures referring to dates in terms of {@code LocalDate}.
193 * <li>Either store the chronology (calendar system) in the user profile or lookup
194 *   the chronology from the user locale
195 * <li>Convert the ISO {@code LocalDate} to and from the user's preferred calendar system during
196 *   printing and parsing
197 * </ul>
198 * This approach treats the problem of globalized calendar systems as a localization issue
199 * and confines it to the UI layer. This approach is in keeping with other localization
200 * issues in the java platform.
201 * <p>
202 * As discussed above, performing calculations on a date where the rules of the calendar system
203 * are pluggable requires skill and is not recommended.
204 * Fortunately, the need to perform calculations on a date in an arbitrary calendar system
205 * is extremely rare. For example, it is highly unlikely that the business rules of a library
206 * book rental scheme will allow rentals to be for one month, where meaning of the month
207 * is dependent on the user's preferred calendar system.
208 * <p>
209 * A key use case for calculations on a date in an arbitrary calendar system is producing
210 * a month-by-month calendar for display and user interaction. Again, this is a UI issue,
211 * and use of this interface solely within a few methods of the UI layer may be justified.
212 * <p>
213 * In any other part of the system, where a date must be manipulated in a calendar system
214 * other than ISO, the use case will generally specify the calendar system to use.
215 * For example, an application may need to calculate the next Islamic or Hebrew holiday
216 * which may require manipulating the date.
217 * This kind of use case can be handled as follows:
218 * <ul>
219 * <li>start from the ISO {@code LocalDate} being passed to the method
220 * <li>convert the date to the alternate calendar system, which for this use case is known
221 *   rather than arbitrary
222 * <li>perform the calculation
223 * <li>convert back to {@code LocalDate}
224 * </ul>
225 * Developers writing low-level frameworks or libraries should also avoid this interface.
226 * Instead, one of the two general purpose access interfaces should be used.
227 * Use {@link TemporalAccessor} if read-only access is required, or use {@link Temporal}
228 * if read-write access is required.
229 *
230 * @implSpec
231 * This interface must be implemented with care to ensure other classes operate correctly.
232 * All implementations that can be instantiated must be final, immutable and thread-safe.
233 * Subclasses should be Serializable wherever possible.
234 * <p>

```



```

235  * Additional calendar systems may be added to the system.
236  * See {@link Chronology} for more details.
237  *
238  * @since 1.8
239  */
240  public interface ChronoLocalDate
241      extends Temporal, TemporalAdjuster, Comparable<ChronoLocalDate> {
242
243      /**
244       * Gets a comparator that compares {@code ChronoLocalDate} in
245       * time-line order ignoring the chronology.
246       * <p>
247       * This comparator differs from the comparison in {@link #compareTo} in that it
248       * only compares the underlying date and not the chronology.
249       * This allows dates in different calendar systems to be compared based
250       * on the position of the date on the local time-line.
251       * The underlying comparison is equivalent to comparing the epoch-day.
252       *
253       * @return a comparator that compares in time-line order ignoring the chronology
254       * @see #isAfter
255       * @see #isBefore
256       * @see #isEqual
257       */
258      static Comparator<ChronoLocalDate> timelineOrder() {
259          return AbstractChronology.DATE_ORDER;
260      }
261
262      //-----
263      /**
264       * Obtains an instance of {@code ChronoLocalDate} from a temporal object.
265       * <p>
266       * This obtains a local date based on the specified temporal.
267       * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
268       * which this factory converts to an instance of {@code ChronoLocalDate}.
269       * <p>
270       * The conversion extracts and combines the chronology and the date
271       * from the temporal object. The behavior is equivalent to using
272       * {@link Chronology#date(TemporalAccessor)} with the extracted chronology.
273       * Implementations are permitted to perform optimizations such as accessing
274       * those fields that are equivalent to the relevant objects.
275       * <p>
276       * This method matches the signature of the functional interface {@link TemporalQuery}
277       * allowing it to be used as a query via method reference, {@code ChronoLocalDate::from}.
278       *
279       * @param temporal the temporal object to convert, not null
280       * @return the date, not null
281       * @throws DateTimeException if unable to convert to a {@code ChronoLocalDate}
282       * @see Chronology#date(TemporalAccessor)
283       */
284      static ChronoLocalDate from(TemporalAccessor temporal) {
285          if (temporal instanceof ChronoLocalDate) {
286              return (ChronoLocalDate) temporal;
287          }

```

```

288     Objects.requireNonNull(temporal, "temporal");
289     Chronology chrono = temporal.query(TemporalQueries.chronology());
290     if (chrono == null) {
291         throw new DateTimeException("Unable to obtain ChronoLocalDate from TemporalAccessor: "
292             + temporal.getClass());
293     }
294     return chrono.date(temporal);
295 }
296
297 //-----
298 /**
299  * Gets the chronology of this date.
300  * <p>
301  * The {@code Chronology} represents the calendar system in use.
302  * The era and other fields in {@link ChronoField} are defined by the chronology.
303  *
304  * @return the chronology, not null
305  */
306 Chronology getChronology();
307
308 /**
309  * Gets the era, as defined by the chronology.
310  * <p>
311  * The era is, conceptually, the largest division of the time-line.
312  * Most calendar systems have a single epoch dividing the time-line into two eras.
313  * However, some have multiple eras, such as one for the reign of each leader.
314  * The exact meaning is determined by the {@code Chronology}.
315  * <p>
316  * All correctly implemented {@code Era} classes are singletons, thus it
317  * is valid code to write {@code date.getEra() == SomeChrono.ERA_NAME)}.
318  * <p>
319  * This default implementation uses {@link Chronology#eraOf(int)}.
320  *
321  * @return the chronology specific era constant applicable at this date, not null
322  */
323 default Era getEra() {
324     return getChronology().eraOf(get(ERA));
325 }
326
327 /**
328  * Checks if the year is a leap year, as defined by the calendar system.
329  * <p>
330  * A leap-year is a year of a longer length than normal.
331  * The exact meaning is determined by the chronology with the constraint that
332  * a leap-year must imply a year-length longer than a non leap-year.
333  * <p>
334  * This default implementation uses {@link Chronology#isLeapYear(long)}.
335  *
336  * @return true if this date is in a leap year, false otherwise
337  */
338 default boolean isLeapYear() {
339     return getChronology().isLeapYear(getLong(YEAR));
340 }

```

```

340
341 /**
342  * Returns the length of the month represented by this date, as defined by the calendar system.
343  * <p>
344  * This returns the length of the month in days.
345  *
346  * @return the length of the month in days
347  */
348 int lengthOfMonth();
349
350 /**
351  * Returns the length of the year represented by this date, as defined by the calendar system.
352  * <p>
353  * This returns the length of the year in days.
354  * <p>
355  * The default implementation uses {@link #isLeapYear()} and returns 365 or 366.
356  *
357  * @return the length of the year in days
358  */
359 default int lengthOfYear() {
360     return (isLeapYear() ? 366 : 365);
361 }
362
363 /**
364  * Checks if the specified field is supported.
365  * <p>
366  * This checks if the specified field can be queried on this date.
367  * If false, then calling the {@link #range(TemporalField) range},
368  * {@link #get(TemporalField) get} and {@link #with(TemporalField, long)}
369  * methods will throw an exception.
370  * <p>
371  * The set of supported fields is defined by the chronology and normally includes
372  * all {@code ChronoField} date fields.
373  * <p>
374  * If the field is not a {@code ChronoField}, then the result of this method
375  * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
376  * passing {@code this} as the argument.
377  * Whether the field is supported is determined by the field.
378  *
379  * @param field the field to check, null returns false
380  * @return true if the field can be queried, false if not
381  */
382 @Override
383 default boolean isSupported(TemporalField field) {
384     if (field instanceof ChronoField) {
385         return field.isDateBased();
386     }
387     return field != null && field.isSupportedBy(this);
388 }
389
390 /**
391  * Checks if the specified unit is supported.
392  * <p>

```

```

393     * This checks if the specified unit can be added to or subtracted from this date.
394     * If false, then calling the {@link #plus(long, TemporalUnit)} and
395     * {@link #minus(long, TemporalUnit) minus} methods will throw an exception.
396     * <p>
397     * The set of supported units is defined by the chronology and normally includes
398     * all {@code ChronoUnit} date units except {@code FOREVER}.
399     * <p>
400     * If the unit is not a {@code ChronoUnit}, then the result of this method
401     * is obtained by invoking {@code TemporalUnit.isSupportedBy(Temporal)}
402     * passing {@code this} as the argument.
403     * Whether the unit is supported is determined by the unit.
404     *
405     * @param unit the unit to check, null returns false
406     * @return true if the unit can be added/subtracted, false if not
407     */
408     @Override
409     default boolean isSupported(TemporalUnit unit) {
410         if (unit instanceof ChronoUnit) {
411             return unit.isDateBased();
412         }
413         return unit != null && unit.isSupportedBy(this);
414     }
415
416     //-----
417     // override for covariant return type
418     /**
419     * {@inheritDoc}
420     * @throws DateTimeException {@inheritDoc}
421     * @throws ArithmeticException {@inheritDoc}
422     */
423     @Override
424     default ChronoLocalDate with(TemporalAdjuster adjuster) {
425         return ChronoLocalDateImpl.ensureValid(getChronology(), Temporal.super.with(adjuster));
426     }
427
428     /**
429     * {@inheritDoc}
430     * @throws DateTimeException {@inheritDoc}
431     * @throws UnsupportedTemporalTypeException {@inheritDoc}
432     * @throws ArithmeticException {@inheritDoc}
433     */
434     @Override
435     default ChronoLocalDate with(TemporalField field, long newValue) {
436         if (field instanceof ChronoField) {
437             throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
438         }
439         return ChronoLocalDateImpl.ensureValid(getChronology(), field.adjustInto(this, newValue));
440     }
441
442     /**
443     * {@inheritDoc}
444     * @throws DateTimeException {@inheritDoc}
445     * @throws ArithmeticException {@inheritDoc}

```

```

446     */
447     @Override
448     default ChronoLocalDate plus(TemporalAmount amount) {
449         return ChronoLocalDateImpl.ensureValid(getChronology(), Temporal.super.plus(amount));
450     }
451
452     /**
453      * {@inheritDoc}
454      * @throws DateTimeException {@inheritDoc}
455      * @throws ArithmeticException {@inheritDoc}
456      */
457     @Override
458     default ChronoLocalDate plus(long amountToAdd, TemporalUnit unit) {
459         if (unit instanceof ChronoUnit) {
460             throw new UnsupportedTemporalTypeException("Unsupported unit: " + unit);
461         }
462         return ChronoLocalDateImpl.ensureValid(getChronology(), unit.addTo(this, amountToAdd));
463     }
464
465     /**
466      * {@inheritDoc}
467      * @throws DateTimeException {@inheritDoc}
468      * @throws ArithmeticException {@inheritDoc}
469      */
470     @Override
471     default ChronoLocalDate minus(TemporalAmount amount) {
472         return ChronoLocalDateImpl.ensureValid(getChronology(), Temporal.super.minus(amount));
473     }
474
475     /**
476      * {@inheritDoc}
477      * @throws DateTimeException {@inheritDoc}
478      * @throws UnsupportedTemporalTypeException {@inheritDoc}
479      * @throws ArithmeticException {@inheritDoc}
480      */
481     @Override
482     default ChronoLocalDate minus(long amountToSubtract, TemporalUnit unit) {
483         return ChronoLocalDateImpl.ensureValid(getChronology(), Temporal.super.minus(
484             amountToSubtract, unit));
485
486     //-----
487     /**
488      * Queries this date using the specified query.
489      * <p>
490      * This queries this date using the specified query strategy object.
491      * The {@code TemporalQuery} object defines the logic to be used to
492      * obtain the result. Read the documentation of the query to understand
493      * what the result of this method will be.
494      * <p>
495      * The result of this method is obtained by invoking the
496      * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
497      * specified query passing {@code this} as the argument.

```

```

498     *
499     * @param <R> the type of the result
500     * @param query the query to invoke, not null
501     * @return the query result, null may be returned (defined by the query)
502     * @throws DateTimeException if unable to query (defined by the query)
503     * @throws ArithmeticException if numeric overflow occurs (defined by the query)
504     */
505     @SuppressWarnings("unchecked")
506     @Override
507     default <R> R query(TemporalQuery<R> query) {
508         if (query == TemporalQueries.zoneId() || query == TemporalQueries.zone() || query ==
            TemporalQueries.offset()) {
509             return null;
510         } else if (query == TemporalQueries.localTime()) {
511             return null;
512         } else if (query == TemporalQueries.chronology()) {
513             return (R) getChronology();
514         } else if (query == TemporalQueries.precision()) {
515             return (R) DAYS;
516         }
517         // inline TemporalAccessor.super.query(query) as an optimization
518         // non-JDK classes are not permitted to make this optimization
519         return query.queryFrom(this);
520     }
521
522     /**
523     * Adjusts the specified temporal object to have the same date as this object.
524     * <p>
525     * This returns a temporal object of the same observable type as the input
526     * with the date changed to be the same as this.
527     * <p>
528     * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
529     * passing {@link ChronoField#EPOCH_DAY} as the field.
530     * <p>
531     * In most cases, it is clearer to reverse the calling pattern by using
532     * {@link Temporal#with(TemporalAdjuster)}:
533     * <pre>
534     * // these two lines are equivalent, but the second approach is recommended
535     * temporal = thisLocalDate.adjustInto(temporal);
536     * temporal = temporal.with(thisLocalDate);
537     * </pre>
538     * <p>
539     * This instance is immutable and unaffected by this method call.
540     *
541     * @param temporal the target object to be adjusted, not null
542     * @return the adjusted object, not null
543     * @throws DateTimeException if unable to make the adjustment
544     * @throws ArithmeticException if numeric overflow occurs
545     */
546     @Override
547     default Temporal adjustInto(Temporal temporal) {
548         return temporal.with(EPOCH_DAY, toEpochDay());
549     }

```

```

550
551 /**
552  * Calculates the amount of time until another date in terms of the specified unit.
553  * <p>
554  * This calculates the amount of time between two {@code ChronoLocalDate}
555  * objects in terms of a single {@code TemporalUnit}.
556  * The start and end points are {@code this} and the specified date.
557  * The result will be negative if the end is before the start.
558  * The {@code Temporal} passed to this method is converted to a
559  * {@code ChronoLocalDate} using {@link Chronology#date(TemporalAccessor)}.
560  * The calculation returns a whole number, representing the number of
561  * complete units between the two dates.
562  * For example, the amount in days between two dates can be calculated
563  * using {@code startDate.until(endDate, DAYS)}.
564  * <p>
565  * There are two equivalent ways of using this method.
566  * The first is to invoke this method.
567  * The second is to use {@link TemporalUnit#between(Temporal, Temporal)}:
568  * <pre>
569  *     // these two lines are equivalent
570  *     amount = start.until(end, MONTHS);
571  *     amount = MONTHS.between(start, end);
572  * </pre>
573  * The choice should be made based on which makes the code more readable.
574  * <p>
575  * The calculation is implemented in this method for {@link ChronoUnit}.
576  * The units {@code DAYS}, {@code WEEKS}, {@code MONTHS}, {@code YEARS},
577  * {@code DECADES}, {@code CENTURIES}, {@code MILLENNIA} and {@code ERAS}
578  * should be supported by all implementations.
579  * Other {@code ChronoUnit} values will throw an exception.
580  * <p>
581  * If the unit is not a {@code ChronoUnit}, then the result of this method
582  * is obtained by invoking {@code TemporalUnit.between(Temporal, Temporal)}
583  * passing {@code this} as the first argument and the converted input temporal as
584  * the second argument.
585  * <p>
586  * This instance is immutable and unaffected by this method call.
587  *
588  * @param endExclusive the end date, exclusive, which is converted to a
589  *     {@code ChronoLocalDate} in the same chronology, not null
590  * @param unit the unit to measure the amount in, not null
591  * @return the amount of time between this date and the end date
592  * @throws DateTimeException if the amount cannot be calculated, or the end
593  *     temporal cannot be converted to a {@code ChronoLocalDate}
594  * @throws UnsupportedTemporalTypeException if the unit is not supported
595  * @throws ArithmeticException if numeric overflow occurs
596  */
597 @Override // override for Javadoc
598 long until(Temporal endExclusive, TemporalUnit unit);
599
600 /**
601  * Calculates the period between this date and another date as a {@code ChronoPeriod}.
602  * <p>

```

```

603     * This calculates the period between two dates. All supplied chronologies
604     * calculate the period using years, months and days, however the
605     * {@code ChronoPeriod} API allows the period to be represented using other units.
606     * <p>
607     * The start and end points are {@code this} and the specified date.
608     * The result will be negative if the end is before the start.
609     * The negative sign will be the same in each of year, month and day.
610     * <p>
611     * The calculation is performed using the chronology of this date.
612     * If necessary, the input date will be converted to match.
613     * <p>
614     * This instance is immutable and unaffected by this method call.
615     *
616     * @param endDateExclusive the end date, exclusive, which may be in any chronology, not null
617     * @return the period between this date and the end date, not null
618     * @throws DateTimeException if the period cannot be calculated
619     * @throws ArithmeticException if numeric overflow occurs
620     */
621     ChronoPeriod until(ChronoLocalDate endDateExclusive);
622
623     /**
624     * Formats this date using the specified formatter.
625     * <p>
626     * This date will be passed to the formatter to produce a string.
627     * <p>
628     * The default implementation must behave as follows:
629     * <pre>
630     * return formatter.format(this);
631     * </pre>
632     *
633     * @param formatter the formatter to use, not null
634     * @return the formatted date string, not null
635     * @throws DateTimeException if an error occurs during printing
636     */
637     default String format(DateTimeFormatter formatter) {
638         Objects.requireNonNull(formatter, "formatter");
639         return formatter.format(this);
640     }
641
642     //-----
643     /**
644     * Combines this date with a time to create a {@code ChronoLocalDateTime}.
645     * <p>
646     * This returns a {@code ChronoLocalDateTime} formed from this date at the specified time.
647     * All possible combinations of date and time are valid.
648     *
649     * @param localTime the local time to use, not null
650     * @return the local date-time formed from this date and the specified time, not null
651     */
652     @SuppressWarnings("unchecked")
653     default ChronoLocalDateTime<?> atTime(LocalTime localTime) {
654         return ChronoLocalDateTimeImpl.of(this, localTime);
655     }

```



```

656
657 //-----
658 /**
659  * Converts this date to the Epoch Day.
660  * <p>
661  * The {@link ChronoField#EPOCH_DAY Epoch Day count} is a simple
662  * incrementing count of days where day 0 is 1970-01-01 (ISO).
663  * This definition is the same for all chronologies, enabling conversion.
664  * <p>
665  * This default implementation queries the {@code EPOCH_DAY} field.
666  *
667  * @return the Epoch Day equivalent to this date
668  */
669 default long toEpochDay() {
670     return getLong(EPOCH_DAY);
671 }
672
673 //-----
674 /**
675  * Compares this date to another date, including the chronology.
676  * <p>
677  * The comparison is based first on the underlying time-line date, then
678  * on the chronology.
679  * It is "consistent with equals", as defined by {@link Comparable}.
680  * <p>
681  * For example, the following is the comparator order:
682  * <ol>
683  * <li>{@code 2012-12-03 (ISO)}</li>
684  * <li>{@code 2012-12-04 (ISO)}</li>
685  * <li>{@code 2555-12-04 (ThaiBuddhist)}</li>
686  * <li>{@code 2012-12-05 (ISO)}</li>
687  * </ol>
688  * Values #2 and #3 represent the same date on the time-line.
689  * When two values represent the same date, the chronology ID is compared to distinguish them.
690  * This step is needed to make the ordering "consistent with equals".
691  * <p>
692  * If all the date objects being compared are in the same chronology, then the
693  * additional chronology stage is not required and only the local date is used.
694  * To compare the dates of two {@code TemporalAccessor} instances, including dates
695  * in two different chronologies, use {@link ChronoField#EPOCH_DAY} as a comparator.
696  * <p>
697  * This default implementation performs the comparison defined above.
698  *
699  * @param other the other date to compare to, not null
700  * @return the comparator value, negative if less, positive if greater
701  */
702 @Override
703 default int compareTo(ChronoLocalDate other) {
704     int cmp = Long.compare(toEpochDay(), other.toEpochDay());
705     if (cmp == 0) {
706         cmp = getChronology().compareTo(other.getChronology());
707     }
708     return cmp;

```

```
709     }
710
711     /**
712      * Checks if this date is after the specified date ignoring the chronology.
713      * <p>
714      * This method differs from the comparison in {@link #compareTo} in that it
715      * only compares the underlying date and not the chronology.
716      * This allows dates in different calendar systems to be compared based
717      * on the time-line position.
718      * This is equivalent to using {@code date1.toEpochDay() > date2.toEpochDay()}.
719      * <p>
720      * This default implementation performs the comparison based on the epoch-day.
721      *
722      * @param other the other date to compare to, not null
723      * @return true if this is after the specified date
724      */
725     default boolean isAfter(ChronoLocalDate other) {
726         return this.toEpochDay() > other.toEpochDay();
727     }
728
729     /**
730      * Checks if this date is before the specified date ignoring the chronology.
731      * <p>
732      * This method differs from the comparison in {@link #compareTo} in that it
733      * only compares the underlying date and not the chronology.
734      * This allows dates in different calendar systems to be compared based
735      * on the time-line position.
736      * This is equivalent to using {@code date1.toEpochDay() < date2.toEpochDay()}.
737      * <p>
738      * This default implementation performs the comparison based on the epoch-day.
739      *
740      * @param other the other date to compare to, not null
741      * @return true if this is before the specified date
742      */
743     default boolean isBefore(ChronoLocalDate other) {
744         return this.toEpochDay() < other.toEpochDay();
745     }
746
747     /**
748      * Checks if this date is equal to the specified date ignoring the chronology.
749      * <p>
750      * This method differs from the comparison in {@link #compareTo} in that it
751      * only compares the underlying date and not the chronology.
752      * This allows dates in different calendar systems to be compared based
753      * on the time-line position.
754      * This is equivalent to using {@code date1.toEpochDay() == date2.toEpochDay()}.
755      * <p>
756      * This default implementation performs the comparison based on the epoch-day.
757      *
758      * @param other the other date to compare to, not null
759      * @return true if the underlying date is equal to the specified date
760      */
761     default boolean isEqual(ChronoLocalDate other) {
```

```

762     return this.toEpochDay() == other.toEpochDay();
763 }
764
765 //-----
766 /**
767  * Checks if this date is equal to another date, including the chronology.
768  * <p>
769  * Compares this date with another ensuring that the date and chronology are the same.
770  * <p>
771  * To compare the dates of two {@code TemporalAccessor} instances, including dates
772  * in two different chronologies, use {@link ChronoField#EPOCH_DAY} as a comparator.
773  *
774  * @param obj the object to check, null returns false
775  * @return true if this is equal to the other date
776  */
777 @Override
778 boolean equals(Object obj);
779
780 /**
781  * A hash code for this date.
782  *
783  * @return a suitable hash code
784  */
785 @Override
786 int hashCode();
787
788 //-----
789 /**
790  * Outputs this date as a {@code String}.
791  * <p>
792  * The output will include the full local date.
793  *
794  * @return the formatted date, not null
795  */
796 @Override
797 String toString();
798
799 }

```

5.3 ChronoLocalDateImpl.java

Secuencia 73: java/time/chrono/ChronoLocalDateImpl.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *

```

```
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
28  *
29  * All rights reserved.
30  *
31  * Redistribution and use in source and binary forms, with or without
32  * modification, are permitted provided that the following conditions are met:
33  *
34  * * Redistributions of source code must retain the above copyright notice,
35  *   this list of conditions and the following disclaimer.
36  *
37  * * Redistributions in binary form must reproduce the above copyright notice,
38  *   this list of conditions and the following disclaimer in the documentation
39  *   and/or other materials provided with the distribution.
40  *
41  * * Neither the name of JSR-310 nor the names of its contributors
42  *   may be used to endorse or promote products derived from this software
43  *   without specific prior written permission.
44  *
45  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56  */
57  package java.time.chrono;
58
59  import static java.time.temporal.ChronoField.DAY_OF_MONTH;
60  import static java.time.temporal.ChronoField.ERA;
61  import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
62  import static java.time.temporal.ChronoField.PROLEPTIC_MONTH;
63  import static java.time.temporal.ChronoField.YEAR_OF_ERA;
64
```

```

65 import java.io.Serializable;
66 import java.time.DateTimeException;
67 import java.time.temporal.ChronoUnit;
68 import java.time.temporal.Temporal;
69 import java.time.temporal.TemporalAdjuster;
70 import java.time.temporal.TemporalAmount;
71 import java.time.temporal.TemporalField;
72 import java.time.temporal.TemporalUnit;
73 import java.time.temporal.UnsupportedTemporalTypeException;
74 import java.time.temporal.ValueRange;
75 import java.util.Objects;
76
77 /**
78  * A date expressed in terms of a standard year-month-day calendar system.
79  * <p>
80  * This class is used by applications seeking to handle dates in non-ISO calendar systems.
81  * For example, the Japanese, Minguo, Thai Buddhist and others.
82  * <p>
83  * {@code ChronoLocalDate} is built on the generic concepts of year, month and day.
84  * The calendar system, represented by a {@link java.time.chrono.Chronology}, expresses the
85  * relationship between
86  * the fields and this class allows the resulting date to be manipulated.
87  * <p>
88  * Note that not all calendar systems are suitable for use with this class.
89  * For example, the Mayan calendar uses a system that bears no relation to years, months and days.
90  * <p>
91  * The API design encourages the use of {@code LocalDate} for the majority of the application.
92  * This includes code to read and write from a persistent data store, such as a database,
93  * and to send dates and times across a network. The {@code ChronoLocalDate} instance is then used
94  * at the user interface level to deal with localized input/output.
95  *
96  * <P>Example: </p>
97  * <pre>
98  *      System.out.printf("Example()\n");
99  *      // Enumerate the list of available calendars and print today for each
100  *      Set<Chronology> chronos = Chronology.getAvailableChronologies();
101  *      for (Chronology chrono : chronos) {
102  *          ChronoLocalDate date = chrono.dateNow();
103  *          System.out.printf("    %20s: %s\n", chrono.getID(), date.toString());
104  *      }
105  *
106  *      // Print the Hijrah date and calendar
107  *      ChronoLocalDate date = Chronology.of("Hijrah").dateNow();
108  *      int day = date.get(ChronoField.DAY_OF_MONTH);
109  *      int dow = date.get(ChronoField.DAY_OF_WEEK);
110  *      int month = date.get(ChronoField.MONTH_OF_YEAR);
111  *      int year = date.get(ChronoField.YEAR);
112  *      System.out.printf("    Today is %s %s %d-%s-%d\n", date.getChronology().getID(),
113  *          dow, day, month, year);
114  *
115  *      // Print today's date and the last day of the year
116  *      ChronoLocalDate now1 = Chronology.of("Hijrah").dateNow();
117  *      ChronoLocalDate first = now1.with(ChronoField.DAY_OF_MONTH, 1)

```

```

117 *         .with(ChronoField.MONTH_OF_YEAR, 1);
118 *         ChronoLocalDate last = first.plus(1, ChronoUnit.YEARS)
119 *         .minus(1, ChronoUnit.DAYS);
120 *         System.out.printf(" Today is %s: start: %s; end: %s%n", last.getChronology().getID(),
121 *         first, last);
122 * </pre>
123 *
124 * <h3>Adding Calendars</h3>
125 * <p> The set of calendars is extensible by defining a subclass of {@link ChronoLocalDate}
126 * to represent a date instance and an implementation of {@code Chronology}
127 * to be the factory for the ChronoLocalDate subclass.
128 * </p>
129 * <p> To permit the discovery of the additional calendar types the implementation of
130 * {@code Chronology} must be registered as a Service implementing the {@code Chronology} interface
131 * in the {@code META-INF/Services} file as per the specification of {@link java.util.ServiceLoader}
132 * .}.
133 * The subclass must function according to the {@code Chronology} class description and must
134 * provide its
135 * {@link java.time.chrono.Chronology#getId() chronology ID} and {@link Chronology#getCalendarType()
136 * calendar type}. </p>
137 *
138 * @implSpec
139 * This abstract class must be implemented with care to ensure other classes operate correctly.
140 * All implementations that can be instantiated must be final, immutable and thread-safe.
141 * Subclasses should be Serializable wherever possible.
142 *
143 * @param <D> the ChronoLocalDate of this date-time
144 * @since 1.8
145 */
146 abstract class ChronoLocalDateImpl<D extends ChronoLocalDate>
147     implements ChronoLocalDate, Temporal, TemporalAdjuster, Serializable {
148
149     /**
150      * Serialization version.
151      */
152     private static final long serialVersionUID = 6282433883239719096L;
153
154     /**
155      * Casts the {@code Temporal} to {@code ChronoLocalDate} ensuring it has the specified
156      * chronology.
157      *
158      * @param chrono the chronology to check for, not null
159      * @param temporal a date-time to cast, not null
160      * @return the date-time checked and cast to {@code ChronoLocalDate}, not null
161      * @throws ClassCastException if the date-time cannot be cast to ChronoLocalDate
162      * or the chronology is not equal this Chronology
163      */
164     static <D extends ChronoLocalDate> D ensureValid(Chronology chrono, Temporal temporal) {
165         @SuppressWarnings("unchecked")
166         D other = (D) temporal;
167         if (chrono.equals(other.getChronology()) == false) {
168             throw new ClassCastException("Chronology mismatch, expected: " + chrono.getId() + ",
169                 actual: " + other.getChronology().getId());
170         }
171     }
172 }

```

```

165     }
166     return other;
167 }
168
169 //-----
170 /**
171  * Creates an instance.
172  */
173 ChronoLocalDateImpl() {
174 }
175
176 @Override
177 @SuppressWarnings("unchecked")
178 public D with(TemporalAdjuster adjuster) {
179     return (D) ChronoLocalDate.super.with(adjuster);
180 }
181
182 @Override
183 @SuppressWarnings("unchecked")
184 public D with(TemporalField field, long value) {
185     return (D) ChronoLocalDate.super.with(field, value);
186 }
187
188 //-----
189 @Override
190 @SuppressWarnings("unchecked")
191 public D plus(TemporalAmount amount) {
192     return (D) ChronoLocalDate.super.plus(amount);
193 }
194
195 //-----
196 @Override
197 @SuppressWarnings("unchecked")
198 public D plus(long amountToAdd, TemporalUnit unit) {
199     if (unit instanceof ChronoUnit) {
200         ChronoUnit f = (ChronoUnit) unit;
201         switch (f) {
202             case DAYS: return plusDays(amountToAdd);
203             case WEEKS: return plusDays(Math.multiplyExact(amountToAdd, 7));
204             case MONTHS: return plusMonths(amountToAdd);
205             case YEARS: return plusYears(amountToAdd);
206             case DECADES: return plusYears(Math.multiplyExact(amountToAdd, 10));
207             case CENTURIES: return plusYears(Math.multiplyExact(amountToAdd, 100));
208             case MILLENNIA: return plusYears(Math.multiplyExact(amountToAdd, 1000));
209             case ERAS: return with(ERA, Math.addExact(getLong(ERA), amountToAdd));
210         }
211         throw new UnsupportedOperationException("Unsupported unit: " + unit);
212     }
213     return (D) ChronoLocalDate.super.plus(amountToAdd, unit);
214 }
215
216 @Override
217 @SuppressWarnings("unchecked")

```

```

218 public D minus(TemporalAmount amount) {
219     return (D) ChronoLocalDate.super.minus(amount);
220 }
221
222 @Override
223 @SuppressWarnings("unchecked")
224 public D minus(long amountToSubtract, TemporalUnit unit) {
225     return (D) ChronoLocalDate.super.minus(amountToSubtract, unit);
226 }
227
228 //-----
229 /**
230  * Returns a copy of this date with the specified number of years added.
231  * <p>
232  * This adds the specified period in years to the date.
233  * In some cases, adding years can cause the resulting date to become invalid.
234  * If this occurs, then other fields, typically the day-of-month, will be adjusted to ensure
235  * that the result is valid. Typically this will select the last valid day of the month.
236  * <p>
237  * This instance is immutable and unaffected by this method call.
238  *
239  * @param yearsToAdd the years to add, may be negative
240  * @return a date based on this one with the years added, not null
241  * @throws DateTimeException if the result exceeds the supported date range
242  */
243 abstract D plusYears(long yearsToAdd);
244
245 /**
246  * Returns a copy of this date with the specified number of months added.
247  * <p>
248  * This adds the specified period in months to the date.
249  * In some cases, adding months can cause the resulting date to become invalid.
250  * If this occurs, then other fields, typically the day-of-month, will be adjusted to ensure
251  * that the result is valid. Typically this will select the last valid day of the month.
252  * <p>
253  * This instance is immutable and unaffected by this method call.
254  *
255  * @param monthsToAdd the months to add, may be negative
256  * @return a date based on this one with the months added, not null
257  * @throws DateTimeException if the result exceeds the supported date range
258  */
259 abstract D plusMonths(long monthsToAdd);
260
261 /**
262  * Returns a copy of this date with the specified number of weeks added.
263  * <p>
264  * This adds the specified period in weeks to the date.
265  * In some cases, adding weeks can cause the resulting date to become invalid.
266  * If this occurs, then other fields will be adjusted to ensure that the result is valid.
267  * <p>
268  * The default implementation uses {@link #plusDays(long)} using a 7 day week.
269  * <p>
270  * This instance is immutable and unaffected by this method call.

```



```

271     *
272     * @param weeksToAdd the weeks to add, may be negative
273     * @return a date based on this one with the weeks added, not null
274     * @throws DateTimeException if the result exceeds the supported date range
275     */
276     D plusWeeks(long weeksToAdd) {
277         return plusDays(Math.multiplyExact(weeksToAdd, 7));
278     }
279
280     /**
281     * Returns a copy of this date with the specified number of days added.
282     * <p>
283     * This adds the specified period in days to the date.
284     * <p>
285     * This instance is immutable and unaffected by this method call.
286     *
287     * @param daysToAdd the days to add, may be negative
288     * @return a date based on this one with the days added, not null
289     * @throws DateTimeException if the result exceeds the supported date range
290     */
291     abstract D plusDays(long daysToAdd);
292
293     //-----
294     /**
295     * Returns a copy of this date with the specified number of years subtracted.
296     * <p>
297     * This subtracts the specified period in years to the date.
298     * In some cases, subtracting years can cause the resulting date to become invalid.
299     * If this occurs, then other fields, typically the day-of-month, will be adjusted to ensure
300     * that the result is valid. Typically this will select the last valid day of the month.
301     * <p>
302     * The default implementation uses {@link #plusYears(long)}.
303     * <p>
304     * This instance is immutable and unaffected by this method call.
305     *
306     * @param yearsToSubtract the years to subtract, may be negative
307     * @return a date based on this one with the years subtracted, not null
308     * @throws DateTimeException if the result exceeds the supported date range
309     */
310     @SuppressWarnings("unchecked")
311     D minusYears(long yearsToSubtract) {
312         return (yearsToSubtract == Long.MIN_VALUE ? ((ChronoLocalDateImpl<D>)plusYears(Long.
313             MAX_VALUE)).plusYears(1) : plusYears(-yearsToSubtract));
314     }
315
316     /**
317     * Returns a copy of this date with the specified number of months subtracted.
318     * <p>
319     * This subtracts the specified period in months to the date.
320     * In some cases, subtracting months can cause the resulting date to become invalid.
321     * If this occurs, then other fields, typically the day-of-month, will be adjusted to ensure
322     * that the result is valid. Typically this will select the last valid day of the month.
323     * <p>

```

```

323     * The default implementation uses {@link #plusMonths(long)}.
324     * <p>
325     * This instance is immutable and unaffected by this method call.
326     *
327     * @param monthsToSubtract the months to subtract, may be negative
328     * @return a date based on this one with the months subtracted, not null
329     * @throws DateTimeException if the result exceeds the supported date range
330     */
331     @SuppressWarnings("unchecked")
332     D minusMonths(long monthsToSubtract) {
333         return (monthsToSubtract == Long.MIN_VALUE ? ((ChronoLocalDateImpl<D>)plusMonths(Long.
334             MAX_VALUE)).plusMonths(1) : plusMonths(-monthsToSubtract));
335     }
336
337     /**
338     * Returns a copy of this date with the specified number of weeks subtracted.
339     * <p>
340     * This subtracts the specified period in weeks to the date.
341     * In some cases, subtracting weeks can cause the resulting date to become invalid.
342     * If this occurs, then other fields will be adjusted to ensure that the result is valid.
343     * <p>
344     * The default implementation uses {@link #plusWeeks(long)}.
345     * <p>
346     * This instance is immutable and unaffected by this method call.
347     *
348     * @param weeksToSubtract the weeks to subtract, may be negative
349     * @return a date based on this one with the weeks subtracted, not null
350     * @throws DateTimeException if the result exceeds the supported date range
351     */
352     @SuppressWarnings("unchecked")
353     D minusWeeks(long weeksToSubtract) {
354         return (weeksToSubtract == Long.MIN_VALUE ? ((ChronoLocalDateImpl<D>)plusWeeks(Long.
355             MAX_VALUE)).plusWeeks(1) : plusWeeks(-weeksToSubtract));
356     }
357
358     /**
359     * Returns a copy of this date with the specified number of days subtracted.
360     * <p>
361     * This subtracts the specified period in days to the date.
362     * <p>
363     * The default implementation uses {@link #plusDays(long)}.
364     * <p>
365     * This instance is immutable and unaffected by this method call.
366     *
367     * @param daysToSubtract the days to subtract, may be negative
368     * @return a date based on this one with the days subtracted, not null
369     * @throws DateTimeException if the result exceeds the supported date range
370     */
371     @SuppressWarnings("unchecked")
372     D minusDays(long daysToSubtract) {
373         return (daysToSubtract == Long.MIN_VALUE ? ((ChronoLocalDateImpl<D>)plusDays(Long.MAX_VALUE
374             )).plusDays(1) : plusDays(-daysToSubtract));
375     }

```

```

373
374 //-----
375 @Override
376 public long until(Temporal endExclusive, TemporalUnit unit) {
377     Objects.requireNonNull(endExclusive, "endExclusive");
378     ChronoLocalDate end = getChronology().date(endExclusive);
379     if (unit instanceof ChronoUnit) {
380         switch ((ChronoUnit) unit) {
381             case DAYS: return daysUntil(end);
382             case WEEKS: return daysUntil(end) / 7;
383             case MONTHS: return monthsUntil(end);
384             case YEARS: return monthsUntil(end) / 12;
385             case DECADES: return monthsUntil(end) / 120;
386             case CENTURIES: return monthsUntil(end) / 1200;
387             case MILLENNIA: return monthsUntil(end) / 12000;
388             case ERAS: return end.getLong(ERA) - getLong(ERA);
389         }
390         throw new UnsupportedOperationException("Unsupported unit: " + unit);
391     }
392     Objects.requireNonNull(unit, "unit");
393     return unit.between(this, end);
394 }
395
396 private long daysUntil(ChronoLocalDate end) {
397     return end.toEpochDay() - toEpochDay(); // no overflow
398 }
399
400 private long monthsUntil(ChronoLocalDate end) {
401     ValueRange range = getChronology().range(MONTH_OF_YEAR);
402     if (range.getMaximum() != 12) {
403         throw new IllegalStateException("ChronoLocalDateImpl only supports Chronologies with 12
404             months per year");
405     }
406     long packed1 = getLong(PROLEPTIC_MONTH) * 32L + get(DAY_OF_MONTH); // no overflow
407     long packed2 = end.getLong(PROLEPTIC_MONTH) * 32L + end.get(DAY_OF_MONTH); // no overflow
408     return (packed2 - packed1) / 32;
409 }
410
411 @Override
412 public boolean equals(Object obj) {
413     if (this == obj) {
414         return true;
415     }
416     if (obj instanceof ChronoLocalDate) {
417         return compareTo((ChronoLocalDate) obj) == 0;
418     }
419     return false;
420 }
421
422 @Override
423 public int hashCode() {
424     long epDay = toEpochDay();
425     return getChronology().hashCode() ^ ((int) (epDay ^ (epDay >>> 32)));

```

```

425     }
426
427     @Override
428     public String toString() {
429         // getLong() reduces chances of exceptions in toString()
430         long yoe = getLong(YEAR_OF_ERA);
431         long moy = getLong(MONTH_OF_YEAR);
432         long dom = getLong(DAY_OF_MONTH);
433         StringBuilder buf = new StringBuilder(30);
434         buf.append(getChronology().toString())
435             .append(" ")
436             .append(getEra())
437             .append(" ")
438             .append(yoe)
439             .append(moy < 10 ? "-0" : "-").append(moy)
440             .append(dom < 10 ? "-0" : "-").append(dom);
441         return buf.toString();
442     }
443
444 }

```

5.4 ChronoLocalDateTime.java

Secuencia 74: java/time/chrono/ChronoLocalDateTime.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *

```

```
30 *
31 *
32 * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
62 package java.time.chrono;
63
64 import static java.time.temporal.ChronoField.EPOCH_DAY;
65 import static java.time.temporal.ChronoField.NANO_OF_DAY;
66 import static java.time.temporal.ChronoUnit.FOREVER;
67 import static java.time.temporal.ChronoUnit.NANOS;
68
69 import java.time.DateTimeException;
70 import java.time.Instant;
71 import java.time.LocalDateTime;
72 import java.time.LocalTime;
73 import java.time.ZoneId;
74 import java.time.ZoneOffset;
75 import java.time.format.DateTimeFormatter;
76 import java.time.temporal.ChronoField;
77 import java.time.temporal.ChronoUnit;
78 import java.time.temporal.Temporal;
79 import java.time.temporal.TemporalAccessor;
80 import java.time.temporal.TemporalAdjuster;
81 import java.time.temporal.TemporalAmount;
82 import java.time.temporal.TemporalField;
```

```

83 import java.time.temporal.TemporalQueries;
84 import java.time.temporal.TemporalQuery;
85 import java.time.temporal.TemporalUnit;
86 import java.time.zone.ZoneRules;
87 import java.util.Comparator;
88 import java.util.Objects;
89
90 /**
91  * A date-time without a time-zone in an arbitrary chronology, intended
92  * for advanced globalization use cases.
93  * <p>
94  * <b>Most applications should declare method signatures, fields and variables
95  * as {@link LocalDateTime}, not this interface.</b>
96  * <p>
97  * A {@code ChronoLocalDateTime} is the abstract representation of a local date-time
98  * where the {@code Chronology chronology}, or calendar system, is pluggable.
99  * The date-time is defined in terms of fields expressed by {@link TemporalField},
100  * where most common implementations are defined in {@link ChronoField}.
101  * The chronology defines how the calendar system operates and the meaning of
102  * the standard fields.
103  *
104  * <h3>When to use this interface</h3>
105  * The design of the API encourages the use of {@code LocalDateTime} rather than this
106  * interface, even in the case where the application needs to deal with multiple
107  * calendar systems. The rationale for this is explored in detail in {@link ChronoLocalDate}.
108  * <p>
109  * Ensure that the discussion in {@code ChronoLocalDate} has been read and understood
110  * before using this interface.
111  *
112  * @implSpec
113  * This interface must be implemented with care to ensure other classes operate correctly.
114  * All implementations that can be instantiated must be final, immutable and thread-safe.
115  * Subclasses should be Serializable wherever possible.
116  *
117  * @param <D> the concrete type for the date of this date-time
118  * @since 1.8
119  */
120 public interface ChronoLocalDateTime<D extends ChronoLocalDate>
121     extends Temporal, TemporalAdjuster, Comparable<ChronoLocalDateTime<?>> {
122
123     /**
124      * Gets a comparator that compares {@code ChronoLocalDateTime} in
125      * time-line order ignoring the chronology.
126      * <p>
127      * This comparator differs from the comparison in {@link #compareTo} in that it
128      * only compares the underlying date-time and not the chronology.
129      * This allows dates in different calendar systems to be compared based
130      * on the position of the date-time on the local time-line.
131      * The underlying comparison is equivalent to comparing the epoch-day and nano-of-day.
132      *
133      * @return a comparator that compares in time-line order ignoring the chronology
134      * @see #isAfter
135      * @see #isBefore

```

```

136     * @see #isEqual
137     */
138     static Comparator<ChronoLocalDateTime<?>> timeLineOrder() {
139         return AbstractChronology.DATE_TIME_ORDER;
140     }
141
142     //-----
143     /**
144      * Obtains an instance of {@code ChronoLocalDateTime} from a temporal object.
145      * <p>
146      * This obtains a local date-time based on the specified temporal.
147      * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
148      * which this factory converts to an instance of {@code ChronoLocalDateTime}.
149      * <p>
150      * The conversion extracts and combines the chronology and the date-time
151      * from the temporal object. The behavior is equivalent to using
152      * {@link Chronology#localDateTime(TemporalAccessor)} with the extracted chronology.
153      * Implementations are permitted to perform optimizations such as accessing
154      * those fields that are equivalent to the relevant objects.
155      * <p>
156      * This method matches the signature of the functional interface {@link TemporalQuery}
157      * allowing it to be used as a query via method reference, {@code ChronoLocalDateTime::from}.
158      *
159      * @param temporal the temporal object to convert, not null
160      * @return the date-time, not null
161      * @throws DateTimeException if unable to convert to a {@code ChronoLocalDateTime}
162      * @see Chronology#localDateTime(TemporalAccessor)
163      */
164     static ChronoLocalDateTime<?> from(TemporalAccessor temporal) {
165         if (temporal instanceof ChronoLocalDateTime) {
166             return (ChronoLocalDateTime<?>) temporal;
167         }
168         Objects.requireNonNull(temporal, "temporal");
169         Chronology chrono = temporal.query(TemporalQueries.chronology());
170         if (chrono == null) {
171             throw new DateTimeException("Unable to obtain ChronoLocalDateTime from TemporalAccessor
172                 : " + temporal.getClass());
173         }
174         return chrono.localDateTime(temporal);
175     }
176     //-----
177     /**
178      * Gets the chronology of this date-time.
179      * <p>
180      * The {@code Chronology} represents the calendar system in use.
181      * The era and other fields in {@link ChronoField} are defined by the chronology.
182      *
183      * @return the chronology, not null
184      */
185     default Chronology getChronology() {
186         return toLocalDate().getChronology();
187     }

```

```

188
189 /**
190  * Gets the local date part of this date-time.
191  * <p>
192  * This returns a local date with the same year, month and day
193  * as this date-time.
194  *
195  * @return the date part of this date-time, not null
196  */
197 D toLocalDate() ;
198
199 /**
200  * Gets the local time part of this date-time.
201  * <p>
202  * This returns a local time with the same hour, minute, second and
203  * nanosecond as this date-time.
204  *
205  * @return the time part of this date-time, not null
206  */
207 LocalTime toLocalTime();
208
209 /**
210  * Checks if the specified field is supported.
211  * <p>
212  * This checks if the specified field can be queried on this date-time.
213  * If false, then calling the {@link #range(TemporalField) range},
214  * {@link #get(TemporalField) get} and {@link #with(TemporalField, long)}
215  * methods will throw an exception.
216  * <p>
217  * The set of supported fields is defined by the chronology and normally includes
218  * all {@code ChronoField} date and time fields.
219  * <p>
220  * If the field is not a {@code ChronoField}, then the result of this method
221  * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
222  * passing {@code this} as the argument.
223  * Whether the field is supported is determined by the field.
224  *
225  * @param field the field to check, null returns false
226  * @return true if the field can be queried, false if not
227  */
228 @Override
229 boolean isSupported(TemporalField field);
230
231 /**
232  * Checks if the specified unit is supported.
233  * <p>
234  * This checks if the specified unit can be added to or subtracted from this date-time.
235  * If false, then calling the {@link #plus(long, TemporalUnit)} and
236  * {@link #minus(long, TemporalUnit) minus} methods will throw an exception.
237  * <p>
238  * The set of supported units is defined by the chronology and normally includes
239  * all {@code ChronoUnit} units except {@code FOREVER}.
240  * <p>

```



```

241     * If the unit is not a {@code ChronoUnit}, then the result of this method
242     * is obtained by invoking {@code TemporalUnit.isSupportedBy(Temporal)}
243     * passing {@code this} as the argument.
244     * Whether the unit is supported is determined by the unit.
245     *
246     * @param unit the unit to check, null returns false
247     * @return true if the unit can be added/subtracted, false if not
248     */
249     @Override
250     default boolean isSupported(TemporalUnit unit) {
251         if (unit instanceof ChronoUnit) {
252             return unit != FOREVER;
253         }
254         return unit != null && unit.isSupportedBy(this);
255     }
256
257     //-----
258     // override for covariant return type
259     /**
260     * {@inheritDoc}
261     * @throws DateTimeException {@inheritDoc}
262     * @throws ArithmeticException {@inheritDoc}
263     */
264     @Override
265     default ChronoLocalDateTime<D> with(TemporalAdjuster adjuster) {
266         return ChronoLocalDateTimeImpl.ensureValid(getChronology(), Temporal.super.with(adjuster));
267     }
268
269     /**
270     * {@inheritDoc}
271     * @throws DateTimeException {@inheritDoc}
272     * @throws ArithmeticException {@inheritDoc}
273     */
274     @Override
275     ChronoLocalDateTime<D> with(TemporalField field, long newValue);
276
277     /**
278     * {@inheritDoc}
279     * @throws DateTimeException {@inheritDoc}
280     * @throws ArithmeticException {@inheritDoc}
281     */
282     @Override
283     default ChronoLocalDateTime<D> plus(TemporalAmount amount) {
284         return ChronoLocalDateTimeImpl.ensureValid(getChronology(), Temporal.super.plus(amount));
285     }
286
287     /**
288     * {@inheritDoc}
289     * @throws DateTimeException {@inheritDoc}
290     * @throws ArithmeticException {@inheritDoc}
291     */
292     @Override
293     ChronoLocalDateTime<D> plus(long amountToAdd, TemporalUnit unit);

```

```

294
295 /**
296  * {@inheritDoc}
297  * @throws DateTimeException {@inheritDoc}
298  * @throws ArithmeticException {@inheritDoc}
299  */
300 @Override
301 default ChronoLocalDateTime<D> minus(TemporalAmount amount) {
302     return ChronoLocalDateTimeImpl.ensureValid(getChronology(), Temporal.super.minus(amount));
303 }
304
305 /**
306  * {@inheritDoc}
307  * @throws DateTimeException {@inheritDoc}
308  * @throws ArithmeticException {@inheritDoc}
309  */
310 @Override
311 default ChronoLocalDateTime<D> minus(long amountToSubtract, TemporalUnit unit) {
312     return ChronoLocalDateTimeImpl.ensureValid(getChronology(), Temporal.super.minus(
313         amountToSubtract, unit));
314 }
315
316 //-----
317 /**
318  * Queries this date-time using the specified query.
319  * <p>
320  * This queries this date-time using the specified query strategy object.
321  * The {@code TemporalQuery} object defines the logic to be used to
322  * obtain the result. Read the documentation of the query to understand
323  * what the result of this method will be.
324  * <p>
325  * The result of this method is obtained by invoking the
326  * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
327  * specified query passing {@code this} as the argument.
328  *
329  * @param <R> the type of the result
330  * @param query the query to invoke, not null
331  * @return the query result, null may be returned (defined by the query)
332  * @throws DateTimeException if unable to query (defined by the query)
333  * @throws ArithmeticException if numeric overflow occurs (defined by the query)
334  */
335 @SuppressWarnings("unchecked")
336 @Override
337 default <R> R query(TemporalQuery<R> query) {
338     if (query == TemporalQueries.zoneId() || query == TemporalQueries.zone() || query ==
339         TemporalQueries.offset()) {
340         return null;
341     } else if (query == TemporalQueries.localTime()) {
342         return (R) toLocalTime();
343     } else if (query == TemporalQueries.chronology()) {
344         return (R) getChronology();
345     } else if (query == TemporalQueries.precision()) {
346         return (R) NANOS;
347     }
348 }

```

```

345     }
346     // inline TemporalAccessor.super.query(query) as an optimization
347     // non-JDK classes are not permitted to make this optimization
348     return query.queryFrom(this);
349 }
350
351 /**
352  * Adjusts the specified temporal object to have the same date and time as this object.
353  * <p>
354  * This returns a temporal object of the same observable type as the input
355  * with the date and time changed to be the same as this.
356  * <p>
357  * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
358  * twice, passing {@link ChronoField#EPOCH_DAY} and
359  * {@link ChronoField#NANO_OF_DAY} as the fields.
360  * <p>
361  * In most cases, it is clearer to reverse the calling pattern by using
362  * {@link Temporal#with(TemporalAdjuster)}:
363  * <pre>
364  * // these two lines are equivalent, but the second approach is recommended
365  * temporal = thisLocalDateTime.adjustInto(temporal);
366  * temporal = temporal.with(thisLocalDateTime);
367  * </pre>
368  * <p>
369  * This instance is immutable and unaffected by this method call.
370  *
371  * @param temporal the target object to be adjusted, not null
372  * @return the adjusted object, not null
373  * @throws DateTimeException if unable to make the adjustment
374  * @throws ArithmeticException if numeric overflow occurs
375  */
376 @Override
377 default Temporal adjustInto(Temporal temporal) {
378     return temporal
379         .with(EPOCH_DAY, toLocalDate().toEpochDay())
380         .with(NANO_OF_DAY, toLocalTime().toNanoOfDay());
381 }
382
383 /**
384  * Formats this date-time using the specified formatter.
385  * <p>
386  * This date-time will be passed to the formatter to produce a string.
387  * <p>
388  * The default implementation must behave as follows:
389  * <pre>
390  * return formatter.format(this);
391  * </pre>
392  *
393  * @param formatter the formatter to use, not null
394  * @return the formatted date-time string, not null
395  * @throws DateTimeException if an error occurs during printing
396  */
397 default String format(DateTimeFormatter formatter) {

```

```

398     Objects.requireNonNull(formatter, "formatter");
399     return formatter.format(this);
400 }
401
402 //-----
403 /**
404  * Combines this time with a time-zone to create a {@code ChronoZonedDateTime}.
405  * <p>
406  * This returns a {@code ChronoZonedDateTime} formed from this date-time at the
407  * specified time-zone. The result will match this date-time as closely as possible.
408  * Time-zone rules, such as daylight savings, mean that not every local date-time
409  * is valid for the specified zone, thus the local date-time may be adjusted.
410  * <p>
411  * The local date-time is resolved to a single instant on the time-line.
412  * This is achieved by finding a valid offset from UTC/Greenwich for the local
413  * date-time as defined by the {@link ZoneRules rules} of the zone ID.
414  * <p>
415  * In most cases, there is only one valid offset for a local date-time.
416  * In the case of an overlap, where clocks are set back, there are two valid offsets.
417  * This method uses the earlier offset typically corresponding to "summer".
418  * <p>
419  * In the case of a gap, where clocks jump forward, there is no valid offset.
420  * Instead, the local date-time is adjusted to be later by the length of the gap.
421  * For a typical one hour daylight savings change, the local date-time will be
422  * moved one hour later into the offset typically corresponding to "summer".
423  * <p>
424  * To obtain the later offset during an overlap, call
425  * {@link ChronoZonedDateTime#withLaterOffsetAtOverlap()} on the result of this method.
426  *
427  * @param zone the time-zone to use, not null
428  * @return the zoned date-time formed from this date-time, not null
429  */
430 ChronoZonedDateTime<D> atZone(ZoneId zone);
431
432 //-----
433 /**
434  * Converts this date-time to an {@code Instant}.
435  * <p>
436  * This combines this local date-time and the specified offset to form
437  * an {@code Instant}.
438  * <p>
439  * This default implementation calculates from the epoch-day of the date and the
440  * second-of-day of the time.
441  *
442  * @param offset the offset to use for the conversion, not null
443  * @return an {@code Instant} representing the same instant, not null
444  */
445 default Instant toInstant(ZoneOffset offset) {
446     return Instant.ofEpochSecond(toEpochSecond(offset), toLocalTime().getNano());
447 }
448
449 /**
450  * Converts this date-time to the number of seconds from the epoch

```

```

451     * of 1970-01-01T00:00:00Z.
452     * <p>
453     * This combines this local date-time and the specified offset to calculate the
454     * epoch-second value, which is the number of elapsed seconds from 1970-01-01T00:00:00Z.
455     * Instants on the time-line after the epoch are positive, earlier are negative.
456     * <p>
457     * This default implementation calculates from the epoch-day of the date and the
458     * second-of-day of the time.
459     *
460     * @param offset the offset to use for the conversion, not null
461     * @return the number of seconds from the epoch of 1970-01-01T00:00:00Z
462     */
463     default long toEpochSecond(ZoneOffset offset) {
464         Objects.requireNonNull(offset, "offset");
465         long epochDay = toLocalDate().toEpochDay();
466         long secs = epochDay * 86400 + toLocalTime().toSecondOfDay();
467         secs -= offset.getTotalSeconds();
468         return secs;
469     }
470
471     //-----
472     /**
473     * Compares this date-time to another date-time, including the chronology.
474     * <p>
475     * The comparison is based first on the underlying time-line date-time, then
476     * on the chronology.
477     * It is "consistent with equals", as defined by {@link Comparable}.
478     * <p>
479     * For example, the following is the comparator order:
480     * <ol>
481     * <li>{@code 2012-12-03T12:00 (ISO)}</li>
482     * <li>{@code 2012-12-04T12:00 (ISO)}</li>
483     * <li>{@code 2555-12-04T12:00 (ThaiBuddhist)}</li>
484     * <li>{@code 2012-12-05T12:00 (ISO)}</li>
485     * </ol>
486     * Values #2 and #3 represent the same date-time on the time-line.
487     * When two values represent the same date-time, the chronology ID is compared to distinguish
488     * them.
489     * This step is needed to make the ordering "consistent with equals".
490     * <p>
491     * If all the date-time objects being compared are in the same chronology, then the
492     * additional chronology stage is not required and only the local date-time is used.
493     * <p>
494     * This default implementation performs the comparison defined above.
495     *
496     * @param other the other date-time to compare to, not null
497     * @return the comparator value, negative if less, positive if greater
498     */
499     @Override
500     default int compareTo(ChronoLocalDateTime<?> other) {
501         int cmp = toLocalDate().compareTo(other.toLocalDate());
502         if (cmp == 0) {
503             cmp = toLocalTime().compareTo(other.toLocalTime());

```

```

503         if (cmp == 0) {
504             cmp = getChronology().compareTo(other.getChronology());
505         }
506     }
507     return cmp;
508 }
509
510 /**
511  * Checks if this date-time is after the specified date-time ignoring the chronology.
512  * <p>
513  * This method differs from the comparison in {@link #compareTo} in that it
514  * only compares the underlying date-time and not the chronology.
515  * This allows dates in different calendar systems to be compared based
516  * on the time-line position.
517  * <p>
518  * This default implementation performs the comparison based on the epoch-day
519  * and nano-of-day.
520  *
521  * @param other the other date-time to compare to, not null
522  * @return true if this is after the specified date-time
523  */
524 default boolean isAfter(ChronoLocalDateTime<?> other) {
525     long thisEpDay = this.toLocalDate().toEpochDay();
526     long otherEpDay = other.toLocalDate().toEpochDay();
527     return thisEpDay > otherEpDay ||
528         (thisEpDay == otherEpDay && this.toLocalTime().toNanoOfDay() > other.toLocalTime().
529             toNanoOfDay());
530 }
531
532 /**
533  * Checks if this date-time is before the specified date-time ignoring the chronology.
534  * <p>
535  * This method differs from the comparison in {@link #compareTo} in that it
536  * only compares the underlying date-time and not the chronology.
537  * This allows dates in different calendar systems to be compared based
538  * on the time-line position.
539  * <p>
540  * This default implementation performs the comparison based on the epoch-day
541  * and nano-of-day.
542  *
543  * @param other the other date-time to compare to, not null
544  * @return true if this is before the specified date-time
545  */
546 default boolean isBefore(ChronoLocalDateTime<?> other) {
547     long thisEpDay = this.toLocalDate().toEpochDay();
548     long otherEpDay = other.toLocalDate().toEpochDay();
549     return thisEpDay < otherEpDay ||
550         (thisEpDay == otherEpDay && this.toLocalTime().toNanoOfDay() < other.toLocalTime().
551             toNanoOfDay());
552 }
553
554 /**
555  * Checks if this date-time is equal to the specified date-time ignoring the chronology.

```

```

554     * <p>
555     * This method differs from the comparison in {@link #compareTo} in that it
556     * only compares the underlying date and time and not the chronology.
557     * This allows date-times in different calendar systems to be compared based
558     * on the time-line position.
559     * <p>
560     * This default implementation performs the comparison based on the epoch-day
561     * and nano-of-day.
562     *
563     * @param other the other date-time to compare to, not null
564     * @return true if the underlying date-time is equal to the specified date-time on the timeline
565     */
566     default boolean isEqual(ChronoLocalDateTime<?> other) {
567         // Do the time check first, it is cheaper than computing EPOCH day.
568         return this.toLocalTime().toNanoOfDay() == other.toLocalTime().toNanoOfDay() &&
569             this.toLocalDate().toEpochDay() == other.toLocalDate().toEpochDay();
570     }
571
572     /**
573     * Checks if this date-time is equal to another date-time, including the chronology.
574     * <p>
575     * Compares this date-time with another ensuring that the date-time and chronology are the same
576     *
577     * @param obj the object to check, null returns false
578     * @return true if this is equal to the other date
579     */
580     @Override
581     boolean equals(Object obj);
582
583     /**
584     * A hash code for this date-time.
585     *
586     * @return a suitable hash code
587     */
588     @Override
589     int hashCode();
590
591     //-----
592     /**
593     * Outputs this date-time as a {@code String}.
594     * <p>
595     * The output will include the full local date-time.
596     *
597     * @return a string representation of this date-time, not null
598     */
599     @Override
600     String toString();
601
602 }

```

Secuencia 75: java/time/chrono/ChronoLocalDateTimeImpl.java

```
1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
```



```

53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.chrono;
63
64  import static java.time.temporal.ChronoField.EPOCH_DAY;
65
66  import java.io.IOException;
67  import java.io.InvalidObjectException;
68  import java.io.ObjectInput;
69  import java.io.ObjectInputStream;
70  import java.io.ObjectOutput;
71  import java.io.Serializable;
72  import java.time.LocalDate;
73  import java.time.ZoneId;
74  import java.time.temporal.ChronoField;
75  import java.time.temporal.ChronoUnit;
76  import java.time.temporal.Temporal;
77  import java.time.temporal.TemporalAdjuster;
78  import java.time.temporal.TemporalField;
79  import java.time.temporal.TemporalUnit;
80  import java.time.temporal.ValueRange;
81  import java.util.Objects;
82
83  /**
84   * A date-time without a time-zone for the calendar neutral API.
85   * <p>
86   * {@code ChronoLocalDateTime} is an immutable date-time object that represents a date-time, often
87   * viewed as year-month-day-hour-minute-second. This object can also access other
88   * fields such as day-of-year, day-of-week and week-of-year.
89   * <p>
90   * This class stores all date and time fields, to a precision of nanoseconds.
91   * It does not store or represent a time-zone. For example, the value
92   * "2nd October 2007 at 13:45.30.123456789" can be stored in an {@code ChronoLocalDateTime}.
93   *
94   * @implSpec
95   * This class is immutable and thread-safe.
96   * @serial
97   * @param <D> the concrete type for the date of this date-time
98   * @since 1.8
99   */
100 final class ChronoLocalDateTimeImpl<D extends ChronoLocalDate>
101     implements ChronoLocalDateTime<D>, Temporal, TemporalAdjuster, Serializable {
102
103     /**
104      * Serialization version.
105      */

```

```
106 private static final long serialVersionUID = 4556003607393004514L;
107 /**
108  * Hours per day.
109  */
110 static final int HOURS_PER_DAY = 24;
111 /**
112  * Minutes per hour.
113  */
114 static final int MINUTES_PER_HOUR = 60;
115 /**
116  * Minutes per day.
117  */
118 static final int MINUTES_PER_DAY = MINUTES_PER_HOUR * HOURS_PER_DAY;
119 /**
120  * Seconds per minute.
121  */
122 static final int SECONDS_PER_MINUTE = 60;
123 /**
124  * Seconds per hour.
125  */
126 static final int SECONDS_PER_HOUR = SECONDS_PER_MINUTE * MINUTES_PER_HOUR;
127 /**
128  * Seconds per day.
129  */
130 static final int SECONDS_PER_DAY = SECONDS_PER_HOUR * HOURS_PER_DAY;
131 /**
132  * Milliseconds per day.
133  */
134 static final long MILLIS_PER_DAY = SECONDS_PER_DAY * 1000L;
135 /**
136  * Microseconds per day.
137  */
138 static final long MICROS_PER_DAY = SECONDS_PER_DAY * 1000_000L;
139 /**
140  * Nanos per second.
141  */
142 static final long NANOS_PER_SECOND = 1000_000_000L;
143 /**
144  * Nanos per minute.
145  */
146 static final long NANOS_PER_MINUTE = NANOS_PER_SECOND * SECONDS_PER_MINUTE;
147 /**
148  * Nanos per hour.
149  */
150 static final long NANOS_PER_HOUR = NANOS_PER_MINUTE * MINUTES_PER_HOUR;
151 /**
152  * Nanos per day.
153  */
154 static final long NANOS_PER_DAY = NANOS_PER_HOUR * HOURS_PER_DAY;
155
156 /**
157  * The date part.
158  */
```

```

159 private final transient D date;
160 /**
161  * The time part.
162  */
163 private final transient LocalTime time;
164
165 //-----
166 /**
167  * Obtains an instance of {@code ChronoLocalDateTime} from a date and time.
168  *
169  * @param date the local date, not null
170  * @param time the local time, not null
171  * @return the local date-time, not null
172  */
173 static <R extends ChronoLocalDate> ChronoLocalDateTimeImpl<R> of(R date, LocalTime time) {
174     return new ChronoLocalDateTimeImpl<>(date, time);
175 }
176
177 /**
178  * Casts the {@code Temporal} to {@code ChronoLocalDateTime} ensuring it has the specified
179  * chronology.
180  *
181  * @param chrono the chronology to check for, not null
182  * @param temporal a date-time to cast, not null
183  * @return the date-time checked and cast to {@code ChronoLocalDateTime}, not null
184  * @throws ClassCastException if the date-time cannot be cast to ChronoLocalDateTimeImpl
185  * or the chronology is not equal this Chronology
186  */
187 static <R extends ChronoLocalDate> ChronoLocalDateTimeImpl<R> ensureValid(Chronology chrono,
188     Temporal temporal) {
189     @SuppressWarnings("unchecked")
190     ChronoLocalDateTimeImpl<R> other = (ChronoLocalDateTimeImpl<R>) temporal;
191     if (chrono.equals(other.getChronology()) == false) {
192         throw new ClassCastException("Chronology mismatch, required: " + chrono.getId()
193             + ", actual: " + other.getChronology().getId());
194     }
195     return other;
196 }
197
198 /**
199  * Constructor.
200  *
201  * @param date the date part of the date-time, not null
202  * @param time the time part of the date-time, not null
203  */
204 private ChronoLocalDateTimeImpl(D date, LocalTime time) {
205     Objects.requireNonNull(date, "date");
206     Objects.requireNonNull(time, "time");
207     this.date = date;
208     this.time = time;
209 }
210
211 /**

```

```

210     * Returns a copy of this date-time with the new date and time, checking
211     * to see if a new object is in fact required.
212     *
213     * @param newDate the date of the new date-time, not null
214     * @param newTime the time of the new date-time, not null
215     * @return the date-time, not null
216     */
217     private ChronoLocalDateTimeImpl<D> with(Temporal newDate, LocalTime newTime) {
218         if (date == newDate && time == newTime) {
219             return this;
220         }
221         // Validate that the new Temporal is a ChronoLocalDate (and not something else)
222         D cd = ChronoLocalDateImpl.ensureValid(date.getChronology(), newDate);
223         return new ChronoLocalDateTimeImpl<>(cd, newTime);
224     }
225
226     //-----
227     @Override
228     public D toLocalDate() {
229         return date;
230     }
231
232     @Override
233     public LocalTime toLocalTime() {
234         return time;
235     }
236
237     //-----
238     @Override
239     public boolean isSupported(TemporalField field) {
240         if (field instanceof ChronoField) {
241             ChronoField f = (ChronoField) field;
242             return f.isDateBased() || f.isTimeBased();
243         }
244         return field != null && field.isSupportedBy(this);
245     }
246
247     @Override
248     public ValueRange range(TemporalField field) {
249         if (field instanceof ChronoField) {
250             ChronoField f = (ChronoField) field;
251             return (f.isTimeBased() ? time.range(field) : date.range(field));
252         }
253         return field.rangeRefinedBy(this);
254     }
255
256     @Override
257     public int get(TemporalField field) {
258         if (field instanceof ChronoField) {
259             ChronoField f = (ChronoField) field;
260             return (f.isTimeBased() ? time.get(field) : date.get(field));
261         }
262         return range(field).checkValidIntValue(getLong(field), field);

```

```

263     }
264
265     @Override
266     public long getLong(TemporalField field) {
267         if (field instanceof ChronoField) {
268             ChronoField f = (ChronoField) field;
269             return (f.isTimeBased() ? time.getLong(field) : date.getLong(field));
270         }
271         return field.getFrom(this);
272     }
273
274     //-----
275     @SuppressWarnings("unchecked")
276     @Override
277     public ChronoLocalDateTimeImpl<D> with(TemporalAdjuster adjuster) {
278         if (adjuster instanceof ChronoLocalDate) {
279             // The Chronology is checked in with(date,time)
280             return with((ChronoLocalDate) adjuster, time);
281         } else if (adjuster instanceof LocalTime) {
282             return with(date, (LocalTime) adjuster);
283         } else if (adjuster instanceof ChronoLocalDateTimeImpl) {
284             return ChronoLocalDateTimeImpl.ensureValid(date.getChronology(), (
285                 ChronoLocalDateTimeImpl<?>) adjuster);
286         }
287         return ChronoLocalDateTimeImpl.ensureValid(date.getChronology(), (ChronoLocalDateTimeImpl
288             <?>) adjuster.adjustInto(this));
289     }
290
291     @Override
292     public ChronoLocalDateTimeImpl<D> with(TemporalField field, long newValue) {
293         if (field instanceof ChronoField) {
294             ChronoField f = (ChronoField) field;
295             if (f.isTimeBased()) {
296                 return with(date, time.with(field, newValue));
297             } else {
298                 return with(date.with(field, newValue), time);
299             }
300             return ChronoLocalDateTimeImpl.ensureValid(date.getChronology(), field.adjustInto(this,
301                 newValue));
302         }
303     }
304
305     //-----
306     @Override
307     public ChronoLocalDateTimeImpl<D> plus(long amountToAdd, TemporalUnit unit) {
308         if (unit instanceof ChronoUnit) {
309             ChronoUnit f = (ChronoUnit) unit;
310             switch (f) {
311                 case NANOS: return plusNanos(amountToAdd);
312                 case MICROS: return plusDays(amountToAdd / MICROS_PER_DAY).plusNanos((amountToAdd %
313                     MICROS_PER_DAY) * 1000);
314                 case MILLIS: return plusDays(amountToAdd / MILLIS_PER_DAY).plusNanos((amountToAdd %
315                     MILLIS_PER_DAY) * 1000000);

```

```

311         case SECONDS: return plusSeconds(amountToAdd);
312         case MINUTES: return plusMinutes(amountToAdd);
313         case HOURS: return plusHours(amountToAdd);
314         case HALF_DAYS: return plusDays(amountToAdd / 256).plusHours((amountToAdd % 256) *
315             12); // no overflow (256 is multiple of 2)
316     }
317     return with(date.plus(amountToAdd, unit), time);
318 }
319 return ChronoLocalDateTimeImpl.ensureValid(date.getChronology(), unit.addTo(this,
320     amountToAdd));
321 }
322
323 private ChronoLocalDateTimeImpl<D> plusDays(long days) {
324     return with(date.plus(days, ChronoUnit.DAYS), time);
325 }
326
327 private ChronoLocalDateTimeImpl<D> plusHours(long hours) {
328     return plusWithOverflow(date, hours, 0, 0, 0);
329 }
330
331 private ChronoLocalDateTimeImpl<D> plusMinutes(long minutes) {
332     return plusWithOverflow(date, 0, minutes, 0, 0);
333 }
334
335 ChronoLocalDateTimeImpl<D> plusSeconds(long seconds) {
336     return plusWithOverflow(date, 0, 0, seconds, 0);
337 }
338
339 private ChronoLocalDateTimeImpl<D> plusNanos(long nanos) {
340     return plusWithOverflow(date, 0, 0, 0, nanos);
341 }
342
343 //-----
344 private ChronoLocalDateTimeImpl<D> plusWithOverflow(D newDate, long hours, long minutes, long
345     seconds, long nanos) {
346     // 9223372036854775808 long, 2147483648 int
347     if ((hours | minutes | seconds | nanos) == 0) {
348         return with(newDate, time);
349     }
350     long totDays = nanos / NANOS_PER_DAY + // max/24*60*60*1B
351         seconds / SECONDS_PER_DAY + // max/24*60*60
352         minutes / MINUTES_PER_DAY + // max/24*60
353         hours / HOURS_PER_DAY; // max/24
354     long totNanos = nanos % NANOS_PER_DAY + // max 8640000000000
355         (seconds % SECONDS_PER_DAY) * NANOS_PER_SECOND + // max 8640000000000
356         (minutes % MINUTES_PER_DAY) * NANOS_PER_MINUTE + // max 8640000000000
357         (hours % HOURS_PER_DAY) * NANOS_PER_HOUR; // max 8640000000000
358     long curNoD = time.toNanoOfDay(); // max 8640000000000
359     totNanos = totNanos + curNoD; // total 43200000000000
360     totDays += Math.floorDiv(totNanos, NANOS_PER_DAY);
361     long newNoD = Math.floorMod(totNanos, NANOS_PER_DAY);
362     LocalTime newTime = (newNoD == curNoD ? time : LocalTime.ofNanoOfDay(newNoD));
363     return with(newDate.plus(totDays, ChronoUnit.DAYS), newTime);

```

```

361     }
362
363     //-----
364     @Override
365     public ChronoZonedDateTime<D> atZone(ZoneId zone) {
366         return ChronoZonedDateTimeImpl.ofBest(this, zone, null);
367     }
368
369     //-----
370     @Override
371     public long until(Temporal endExclusive, TemporalUnit unit) {
372         Objects.requireNonNull(endExclusive, "endExclusive");
373         @SuppressWarnings("unchecked")
374         ChronoLocalDateTime<D> end = (ChronoLocalDateTime<D>) getChronology().localDateTime(
375             endExclusive);
376         if (unit instanceof ChronoUnit) {
377             if (unit.isTimeBased()) {
378                 long amount = end.getLong(EPOCH_DAY) - date.getLong(EPOCH_DAY);
379                 switch ((ChronoUnit) unit) {
380                     case NANOS: amount = Math.multiplyExact(amount, NANOS_PER_DAY); break;
381                     case MICROS: amount = Math.multiplyExact(amount, MICROS_PER_DAY); break;
382                     case MILLIS: amount = Math.multiplyExact(amount, MILLIS_PER_DAY); break;
383                     case SECONDS: amount = Math.multiplyExact(amount, SECONDS_PER_DAY); break;
384                     case MINUTES: amount = Math.multiplyExact(amount, MINUTES_PER_DAY); break;
385                     case HOURS: amount = Math.multiplyExact(amount, HOURS_PER_DAY); break;
386                     case HALF_DAYS: amount = Math.multiplyExact(amount, 2); break;
387                 }
388                 return Math.addExact(amount, time.until(end.toLocalTime(), unit));
389             }
390             ChronoLocalDate endDate = end.toLocalDate();
391             if (end.toLocalTime().isBefore(time)) {
392                 endDate = endDate.minus(1, ChronoUnit.DAYS);
393             }
394             return date.until(endDate, unit);
395         }
396         Objects.requireNonNull(unit, "unit");
397         return unit.between(this, end);
398     }
399
400     //-----
401     /**
402     * Writes the ChronoLocalDateTime using a
403     * <a href="../../serialized-form.html#java.time.chrono.Ser">dedicated serialized form</a>.
404     * @serialData
405     * <pre>
406     *   out.writeByte(2);           // identifies a ChronoLocalDateTime
407     *   out.writeObject(toLocalDate());
408     *   out.writeObject(toLocalTime());
409     * </pre>
410     *
411     * @return the instance of {@code Ser}, not null
412     */
413     private Object writeReplace() {

```

```

413     return new Ser(Ser.CHRONO_LOCAL_DATE_TIME_TYPE, this);
414 }
415
416 /**
417  * Defend against malicious streams.
418  *
419  * @param s the stream to read
420  * @throws InvalidObjectException always
421  */
422 private void readObject(ObjectInputStream s) throws InvalidObjectException {
423     throw new InvalidObjectException("Deserialization via serialization delegate");
424 }
425
426 void writeExternal(ObjectOutput out) throws IOException {
427     out.writeObject(date);
428     out.writeObject(time);
429 }
430
431 static ChronoLocalDateTime<?> readExternal(ObjectInput in) throws IOException,
432     ClassNotFoundException {
433     ChronoLocalDate date = (ChronoLocalDate) in.readObject();
434     LocalDateTime time = (LocalTime) in.readObject();
435     return date.atTime(time);
436 }
437 //-----
438 @Override
439 public boolean equals(Object obj) {
440     if (this == obj) {
441         return true;
442     }
443     if (obj instanceof ChronoLocalDateTime) {
444         return compareTo((ChronoLocalDateTime<?>) obj) == 0;
445     }
446     return false;
447 }
448
449 @Override
450 public int hashCode() {
451     return toLocalDate().hashCode() ^ toLocalTime().hashCode();
452 }
453
454 @Override
455 public String toString() {
456     return toLocalDate().toString() + 'T' + toLocalTime().toString();
457 }
458
459 }

```

5.6 ChronoPeriod.java

Secuencia 76: java/time/chrono/ChronoPeriod.java

```
1  /*
```



```
2  * Copyright (c) 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2013, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
```

```

55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.chrono;
63
64  import java.time.DateTimeException;
65  import java.time.temporal.ChronoUnit;
66  import java.time.temporal.Temporal;
67  import java.time.temporal.TemporalAmount;
68  import java.time.temporal.TemporalUnit;
69  import java.time.temporal.UnsupportedTemporalTypeException;
70  import java.util.List;
71  import java.util.Objects;
72
73  /**
74   * A date-based amount of time, such as '3 years, 4 months and 5 days' in an
75   * arbitrary chronology, intended for advanced globalization use cases.
76   * <p>
77   * This interface models a date-based amount of time in a calendar system.
78   * While most calendar systems use years, months and days, some do not.
79   * Therefore, this interface operates solely in terms of a set of supported
80   * units that are defined by the {@code Chronology}.
81   * The set of supported units is fixed for a given chronology.
82   * The amount of a supported unit may be set to zero.
83   * <p>
84   * The period is modeled as a directed amount of time, meaning that individual
85   * parts of the period may be negative.
86   *
87   * @implSpec
88   * This interface must be implemented with care to ensure other classes operate correctly.
89   * All implementations that can be instantiated must be final, immutable and thread-safe.
90   * Subclasses should be Serializable wherever possible.
91   *
92   * @since 1.8
93   */
94  public interface ChronoPeriod
95      extends TemporalAmount {
96
97      /**
98       * Obtains a {@code ChronoPeriod} consisting of amount of time between two dates.
99       * <p>
100      * The start date is included, but the end date is not.
101      * The period is calculated using {@link ChronoLocalDate#until(ChronoLocalDate)}.
102      * As such, the calculation is chronology specific.
103      * <p>
104      * The chronology of the first date is used.
105      * The chronology of the second date is ignored, with the date being converted
106      * to the target chronology system before the calculation starts.
107      * <p>

```

```

108     * The result of this method can be a negative period if the end is before the start.
109     * In most cases, the positive/negative sign will be the same in each of the supported fields.
110     *
111     * @param startDateInclusive the start date, inclusive, specifying the chronology of the
112         calculation, not null
113     * @param endDateExclusive the end date, exclusive, in any chronology, not null
114     * @return the period between this date and the end date, not null
115     * @see ChronoLocalDate#until(ChronoLocalDate)
116     */
117     public static ChronoPeriod between(ChronoLocalDate startDateInclusive, ChronoLocalDate
118         endDateExclusive) {
119         Objects.requireNonNull(startDateInclusive, "startDateInclusive");
120         Objects.requireNonNull(endDateExclusive, "endDateExclusive");
121         return startDateInclusive.until(endDateExclusive);
122     }
123
124     //-----
125     /**
126     * Gets the value of the requested unit.
127     * <p>
128     * The supported units are chronology specific.
129     * They will typically be {@link ChronoUnit#YEARS YEARS},
130     * {@link ChronoUnit#MONTHS MONTHS} and {@link ChronoUnit#DAYS DAYS}.
131     * Requesting an unsupported unit will throw an exception.
132     *
133     * @param unit the {@code TemporalUnit} for which to return the value
134     * @return the long value of the unit
135     * @throws DateTimeException if the unit is not supported
136     * @throws UnsupportedTemporalTypeException if the unit is not supported
137     */
138     @Override
139     long get(TemporalUnit unit);
140
141     /**
142     * Gets the set of units supported by this period.
143     * <p>
144     * The supported units are chronology specific.
145     * They will typically be {@link ChronoUnit#YEARS YEARS},
146     * {@link ChronoUnit#MONTHS MONTHS} and {@link ChronoUnit#DAYS DAYS}.
147     * They are returned in order from largest to smallest.
148     * <p>
149     * This set can be used in conjunction with {@link #get(TemporalUnit)}
150     * to access the entire state of the period.
151     *
152     * @return a list containing the supported units, not null
153     */
154     @Override
155     List<TemporalUnit> getUnits();
156
157     /**
158     * Gets the chronology that defines the meaning of the supported units.
159     * <p>
160     * The period is defined by the chronology.

```

```

159     * It controls the supported units and restricts addition/subtraction
160     * to {@code ChronoLocalDate} instances of the same chronology.
161     *
162     * @return the chronology defining the period, not null
163     */
164     Chronology getChronology();
165
166     //-----
167     /**
168      * Checks if all the supported units of this period are zero.
169      *
170      * @return true if this period is zero-length
171      */
172     default boolean isZero() {
173         for (TemporalUnit unit : getUnits()) {
174             if (get(unit) != 0) {
175                 return false;
176             }
177         }
178         return true;
179     }
180
181     /**
182      * Checks if any of the supported units of this period are negative.
183      *
184      * @return true if any unit of this period is negative
185      */
186     default boolean isNegative() {
187         for (TemporalUnit unit : getUnits()) {
188             if (get(unit) < 0) {
189                 return true;
190             }
191         }
192         return false;
193     }
194
195     //-----
196     /**
197      * Returns a copy of this period with the specified period added.
198      * <p>
199      * If the specified amount is a {@code ChronoPeriod} then it must have
200      * the same chronology as this period. Implementations may choose to
201      * accept or reject other {@code TemporalAmount} implementations.
202      * <p>
203      * This instance is immutable and unaffected by this method call.
204      *
205      * @param amountToAdd the period to add, not null
206      * @return a {@code ChronoPeriod} based on this period with the requested period added, not
207      *         null
208      * @throws ArithmeticException if numeric overflow occurs
209      */
210     ChronoPeriod plus(TemporalAmount amountToAdd);

```

```

211 /**
212  * Returns a copy of this period with the specified period subtracted.
213  * <p>
214  * If the specified amount is a {@code ChronoPeriod} then it must have
215  * the same chronology as this period. Implementations may choose to
216  * accept or reject other {@code TemporalAmount} implementations.
217  * <p>
218  * This instance is immutable and unaffected by this method call.
219  *
220  * @param amountToSubtract the period to subtract, not null
221  * @return a {@code ChronoPeriod} based on this period with the requested period subtracted,
222  *         not null
223  * @throws ArithmeticException if numeric overflow occurs
224  */
225 ChronoPeriod minus(TemporalAmount amountToSubtract);
226
227 //-----
228 /**
229  * Returns a new instance with each amount in this period in this period
230  * multiplied by the specified scalar.
231  * <p>
232  * This returns a period with each supported unit individually multiplied.
233  * For example, a period of "2 years, -3 months and 4 days" multiplied by
234  * 3 will return "6 years, -9 months and 12 days".
235  * No normalization is performed.
236  *
237  * @param scalar the scalar to multiply by, not null
238  * @return a {@code ChronoPeriod} based on this period with the amounts multiplied
239  *         by the scalar, not null
240  * @throws ArithmeticException if numeric overflow occurs
241  */
242 ChronoPeriod multipliedBy(int scalar);
243
244 /**
245  * Returns a new instance with each amount in this period negated.
246  * <p>
247  * This returns a period with each supported unit individually negated.
248  * For example, a period of "2 years, -3 months and 4 days" will be
249  * negated to "-2 years, 3 months and -4 days".
250  * No normalization is performed.
251  *
252  * @return a {@code ChronoPeriod} based on this period with the amounts negated, not null
253  * @throws ArithmeticException if numeric overflow occurs, which only happens if
254  *         one of the units has the value {@code Long.MIN_VALUE}
255  */
256 default ChronoPeriod negated() {
257     return multipliedBy(-1);
258 }
259
260 //-----
261 /**
262  * Returns a copy of this period with the amounts of each unit normalized.
263  * <p>

```

```

263     * The process of normalization is specific to each calendar system.
264     * For example, in the ISO calendar system, the years and months are
265     * normalized but the days are not, such that "15 months" would be
266     * normalized to "1 year and 3 months".
267     * <p>
268     * This instance is immutable and unaffected by this method call.
269     *
270     * @return a {@code ChronoPeriod} based on this period with the amounts of each
271     * unit normalized, not null
272     * @throws ArithmeticException if numeric overflow occurs
273     */
274     ChronoPeriod normalized();
275
276     //-----
277     /**
278     * Adds this period to the specified temporal object.
279     * <p>
280     * This returns a temporal object of the same observable type as the input
281     * with this period added.
282     * <p>
283     * In most cases, it is clearer to reverse the calling pattern by using
284     * {@link Temporal#plus(TemporalAmount)}.
285     * <pre>
286     * // these two lines are equivalent, but the second approach is recommended
287     * dateTime = thisPeriod.addTo(dateTime);
288     * dateTime = dateTime.plus(thisPeriod);
289     * </pre>
290     * <p>
291     * The specified temporal must have the same chronology as this period.
292     * This returns a temporal with the non-zero supported units added.
293     * <p>
294     * This instance is immutable and unaffected by this method call.
295     *
296     * @param temporal the temporal object to adjust, not null
297     * @return an object of the same type with the adjustment made, not null
298     * @throws DateTimeException if unable to add
299     * @throws ArithmeticException if numeric overflow occurs
300     */
301     @Override
302     Temporal addTo(Temporal temporal);
303
304     /**
305     * Subtracts this period from the specified temporal object.
306     * <p>
307     * This returns a temporal object of the same observable type as the input
308     * with this period subtracted.
309     * <p>
310     * In most cases, it is clearer to reverse the calling pattern by using
311     * {@link Temporal#minus(TemporalAmount)}.
312     * <pre>
313     * // these two lines are equivalent, but the second approach is recommended
314     * dateTime = thisPeriod.subtractFrom(dateTime);
315     * dateTime = dateTime.minus(thisPeriod);

```

```

316     * </pre>
317     * <p>
318     * The specified temporal must have the same chronology as this period.
319     * This returns a temporal with the non-zero supported units subtracted.
320     * <p>
321     * This instance is immutable and unaffected by this method call.
322     *
323     * @param temporal the temporal object to adjust, not null
324     * @return an object of the same type with the adjustment made, not null
325     * @throws DateTimeException if unable to subtract
326     * @throws ArithmeticException if numeric overflow occurs
327     */
328     @Override
329     Temporal subtractFrom(Temporal temporal);
330
331     //-----
332     /**
333     * Checks if this period is equal to another period, including the chronology.
334     * <p>
335     * Compares this period with another ensuring that the type, each amount and
336     * the chronology are the same.
337     * Note that this means that a period of "15 Months" is not equal to a period
338     * of "1 Year and 3 Months".
339     *
340     * @param obj the object to check, null returns false
341     * @return true if this is equal to the other period
342     */
343     @Override
344     boolean equals(Object obj);
345
346     /**
347     * A hash code for this period.
348     *
349     * @return a suitable hash code
350     */
351     @Override
352     int hashCode();
353
354     //-----
355     /**
356     * Outputs this period as a {@code String}.
357     * <p>
358     * The output will include the period amounts and chronology.
359     *
360     * @return a string representation of this period, not null
361     */
362     @Override
363     String toString();
364
365 }

```

Secuencia 77: java/time/chrono/ChronoPeriodImpl.java

```
1  /*
2  * Copyright (c) 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * Copyright (c) 2013, Stephen Colebourne & Michael Nascimento Santos
28 *
29 * All rights reserved.
30 *
31 * Redistribution and use in source and binary forms, with or without
32 * modification, are permitted provided that the following conditions are met:
33 *
34 * * Redistributions of source code must retain the above copyright notice,
35 *   this list of conditions and the following disclaimer.
36 *
37 * * Redistributions in binary form must reproduce the above copyright notice,
38 *   this list of conditions and the following disclaimer in the documentation
39 *   and/or other materials provided with the distribution.
40 *
41 * * Neither the name of JSR-310 nor the names of its contributors
42 *   may be used to endorse or promote products derived from this software
43 *   without specific prior written permission.
44 *
45 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
```



```

53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56  */
57  package java.time.chrono;
58
59  import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
60  import static java.time.temporal.ChronoUnit.DAYS;
61  import static java.time.temporal.ChronoUnit.MONTHS;
62  import static java.time.temporal.ChronoUnit.YEARS;
63
64  import java.io.DataInput;
65  import java.io.DataOutput;
66  import java.io.IOException;
67  import java.io.InvalidObjectException;
68  import java.io.ObjectInputStream;
69  import java.io.ObjectStreamException;
70  import java.io.Serializable;
71  import java.time.DateTimeException;
72  import java.time.temporal.ChronoUnit;
73  import java.time.temporal.Temporal;
74  import java.time.temporal.TemporalAccessor;
75  import java.time.temporal.TemporalAmount;
76  import java.time.temporal.TemporalQueries;
77  import java.time.temporal.TemporalUnit;
78  import java.time.temporal.UnsupportedTemporalTypeException;
79  import java.time.temporal.ValueRange;
80  import java.util.Arrays;
81  import java.util.Collections;
82  import java.util.List;
83  import java.util.Objects;
84
85  /**
86   * A period expressed in terms of a standard year-month-day calendar system.
87   * <p>
88   * This class is used by applications seeking to handle dates in non-ISO calendar systems.
89   * For example, the Japanese, Minguo, Thai Buddhist and others.
90   *
91   * @implSpec
92   * This class is immutable nad thread-safe.
93   *
94   * @since 1.8
95   */
96  final class ChronoPeriodImpl
97      implements ChronoPeriod, Serializable {
98      // this class is only used by JDK chronology implementations and makes assumptions based on
99      // that fact
100
101      /**
102       * Serialization version.
103       */
104      private static final long serialVersionUID = 57387258289L;

```

```

105  /**
106   * The set of supported units.
107   */
108  private static final List<TemporalUnit> SUPPORTED_UNITS =
109      Collections.unmodifiableList(Arrays.<TemporalUnit>asList(YEARS, MONTHS, DAYS));
110
111  /**
112   * The chronology.
113   */
114  private final Chronology chrono;
115  /**
116   * The number of years.
117   */
118  final int years;
119  /**
120   * The number of months.
121   */
122  final int months;
123  /**
124   * The number of days.
125   */
126  final int days;
127
128  /**
129   * Creates an instance.
130   */
131  ChronoPeriodImpl(Chronology chrono, int years, int months, int days) {
132      Objects.requireNonNull(chono, "chono");
133      this.chono = chrono;
134      this.years = years;
135      this.months = months;
136      this.days = days;
137  }
138
139  //-----
140  @Override
141  public long get(TemporalUnit unit) {
142      if (unit == ChronoUnit.YEARS) {
143          return years;
144      } else if (unit == ChronoUnit.MONTHS) {
145          return months;
146      } else if (unit == ChronoUnit.DAYS) {
147          return days;
148      } else {
149          throw new UnsupportedOperationException("Unsupported unit: " + unit);
150      }
151  }
152
153  @Override
154  public List<TemporalUnit> getUnits() {
155      return ChronoPeriodImpl.SUPPORTED_UNITS;
156  }
157

```

```

158     @Override
159     public Chronology getChronology() {
160         return chrono;
161     }
162
163     //-----
164     @Override
165     public boolean isZero() {
166         return years == 0 && months == 0 && days == 0;
167     }
168
169     @Override
170     public boolean isNegative() {
171         return years < 0 || months < 0 || days < 0;
172     }
173
174     //-----
175     @Override
176     public ChronoPeriod plus(TemporalAmount amountToAdd) {
177         ChronoPeriodImpl amount = validateAmount(amountToAdd);
178         return new ChronoPeriodImpl(
179             chrono,
180             Math.addExact(years, amount.years),
181             Math.addExact(months, amount.months),
182             Math.addExact(days, amount.days));
183     }
184
185     @Override
186     public ChronoPeriod minus(TemporalAmount amountToSubtract) {
187         ChronoPeriodImpl amount = validateAmount(amountToSubtract);
188         return new ChronoPeriodImpl(
189             chrono,
190             Math.subtractExact(years, amount.years),
191             Math.subtractExact(months, amount.months),
192             Math.subtractExact(days, amount.days));
193     }
194
195     /**
196      * Obtains an instance of {@code ChronoPeriodImpl} from a temporal amount.
197      *
198      * @param amount the temporal amount to convert, not null
199      * @return the period, not null
200      */
201     private ChronoPeriodImpl validateAmount(TemporalAmount amount) {
202         Objects.requireNonNull(amount, "amount");
203         if (amount instanceof ChronoPeriodImpl == false) {
204             throw new DateTimeException("Unable to obtain ChronoPeriod from TemporalAmount: " +
205                 amount.getClass());
206         }
207         ChronoPeriodImpl period = (ChronoPeriodImpl) amount;
208         if (chrono.equals(period.getChronology()) == false) {
209             throw new ClassCastException("Chronology mismatch, expected: " + chrono.getId() + ",
210                 actual: " + period.getChronology().getId());

```

```

209     }
210     return period;
211 }
212
213 //-----
214 @Override
215 public ChronoPeriod multipliedBy(int scalar) {
216     if (this.isZero() || scalar == 1) {
217         return this;
218     }
219     return new ChronoPeriodImpl(
220         chrono,
221         Math.multiplyExact(years, scalar),
222         Math.multiplyExact(months, scalar),
223         Math.multiplyExact(days, scalar));
224 }
225
226 //-----
227 @Override
228 public ChronoPeriod normalized() {
229     long monthRange = monthRange();
230     if (monthRange > 0) {
231         long totalMonths = years * monthRange + months;
232         long splitYears = totalMonths / monthRange;
233         int splitMonths = (int) (totalMonths % monthRange); // no overflow
234         if (splitYears == years && splitMonths == months) {
235             return this;
236         }
237         return new ChronoPeriodImpl(chrono, Math.toIntExact(splitYears), splitMonths, days);
238     }
239     return this;
240 }
241
242
243 /**
244  * Calculates the range of months.
245  *
246  * @return the month range, -1 if not fixed range
247  */
248 private long monthRange() {
249     ValueRange startRange = chrono.range(MONTH_OF_YEAR);
250     if (startRange.isFixed() && startRange.isIntValue()) {
251         return startRange.getMaximum() - startRange.getMinimum() + 1;
252     }
253     return -1;
254 }
255
256 //-----
257 @Override
258 public Temporal addTo(Temporal temporal) {
259     validateChrono(temporal);
260     if (months == 0) {
261         if (years != 0) {

```

```
262         temporal = temporal.plus(years, YEARS);
263     }
264     } else {
265         long monthRange = monthRange();
266         if (monthRange > 0) {
267             temporal = temporal.plus(years * monthRange + months, MONTHS);
268         } else {
269             if (years != 0) {
270                 temporal = temporal.plus(years, YEARS);
271             }
272             temporal = temporal.plus(months, MONTHS);
273         }
274     }
275     if (days != 0) {
276         temporal = temporal.plus(days, DAYS);
277     }
278     return temporal;
279 }
280
281
282
283 @Override
284 public Temporal subtractFrom(Temporal temporal) {
285     validateChrono(temporal);
286     if (months == 0) {
287         if (years != 0) {
288             temporal = temporal.minus(years, YEARS);
289         }
290     } else {
291         long monthRange = monthRange();
292         if (monthRange > 0) {
293             temporal = temporal.minus(years * monthRange + months, MONTHS);
294         } else {
295             if (years != 0) {
296                 temporal = temporal.minus(years, YEARS);
297             }
298             temporal = temporal.minus(months, MONTHS);
299         }
300     }
301     if (days != 0) {
302         temporal = temporal.minus(days, DAYS);
303     }
304     return temporal;
305 }
306
307 /**
308  * Validates that the temporal has the correct chronology.
309  */
310 private void validateChrono(TemporalAccessor temporal) {
311     Objects.requireNonNull(temporal, "temporal");
312     Chronology temporalChrono = temporal.query(TemporalQueries.chronology());
313     if (temporalChrono != null && chrono.equals(temporalChrono) == false) {
314         throw new DateTimeException("Chronology mismatch, expected: " + chrono.getId() + ",
```

```

        actual: " + temporalChrono.getId());
    }
}

//-----
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj instanceof ChronoPeriodImpl) {
        ChronoPeriodImpl other = (ChronoPeriodImpl) obj;
        return years == other.years && months == other.months &&
            days == other.days && chrono.equals(other.chrono);
    }
    return false;
}

@Override
public int hashCode() {
    return (years + Integer.rotateLeft(months, 8) + Integer.rotateLeft(days, 16)) ^ chrono.
        hashCode();
}

//-----
@Override
public String toString() {
    if (isZero()) {
        return getChronology().toString() + " POD";
    } else {
        StringBuilder buf = new StringBuilder();
        buf.append(getChronology().toString()).append(' ').append('P');
        if (years != 0) {
            buf.append(years).append('Y');
        }
        if (months != 0) {
            buf.append(months).append('M');
        }
        if (days != 0) {
            buf.append(days).append('D');
        }
        return buf.toString();
    }
}

//-----
/**
 * Writes the Chronology using a
 * <a href="../../serialized-form.html#java.time.chrono.Ser">dedicated serialized form</a>.
 * <pre>
 * out.writeByte(12); // identifies this as a ChronoPeriodImpl
 * out.writeUTF(getId()); // the chronology
 * out.writeInt(years);

```

```

366     * out.writeInt(months);
367     * out.writeInt(days);
368     * </pre>
369     *
370     * @return the instance of {@code Ser}, not null
371     */
372     protected Object writeReplace() {
373         return new Ser(Ser.CHRONO_PERIOD_TYPE, this);
374     }
375
376     /**
377     * Defend against malicious streams.
378     *
379     * @param s the stream to read
380     * @throws InvalidObjectException always
381     */
382     private void readObject(ObjectInputStream s) throws ObjectStreamException {
383         throw new InvalidObjectException("Deserialization via serialization delegate");
384     }
385
386     void writeExternal(DataOutput out) throws IOException {
387         out.writeUTF(chrono.getId());
388         out.writeInt(years);
389         out.writeInt(months);
390         out.writeInt(days);
391     }
392
393     static ChronoPeriodImpl readExternal(DataInput in) throws IOException {
394         Chronology chrono = Chronology.of(in.readUTF());
395         int years = in.readInt();
396         int months = in.readInt();
397         int days = in.readInt();
398         return new ChronoPeriodImpl(chrono, years, months, days);
399     }
400
401 }

```

5.8 ChronoZonedDateTime.java

Secuencia 78: java/time/chrono/ChronoZonedDateTime.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *

```

```
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.chrono;
63
64  import static java.time.temporal.ChronoField.INSTANT_SECONDS;
65  import static java.time.temporal.ChronoField.OFFSET_SECONDS;
66  import static java.time.temporal.ChronoUnit.FOREVER;
```



```
67 import static java.time.temporal.ChronoUnit.NANOS;
68
69 import java.time.DateTimeException;
70 import java.time.Instant;
71 import java.time.LocalDate;
72 import java.time.ZoneId;
73 import java.time.ZoneOffset;
74 import java.time.ZonedDateTime;
75 import java.time.format.DateTimeFormatter;
76 import java.time.temporal.ChronoField;
77 import java.time.temporal.ChronoUnit;
78 import java.time.temporal.Temporal;
79 import java.time.temporal.TemporalAccessor;
80 import java.time.temporal.TemporalAdjuster;
81 import java.time.temporal.TemporalAmount;
82 import java.time.temporal.TemporalField;
83 import java.time.temporal.TemporalQueries;
84 import java.time.temporal.TemporalQuery;
85 import java.time.temporal.TemporalUnit;
86 import java.time.temporal.UnsupportedTemporalTypeException;
87 import java.time.temporal.ValueRange;
88 import java.util.Comparator;
89 import java.util.Objects;
90
91 /**
92  * A date-time with a time-zone in an arbitrary chronology,
93  * intended for advanced globalization use cases.
94  * <p>
95  * <b>Most applications should declare method signatures, fields and variables
96  * as {@link ZonedDateTime}, not this interface.</b>
97  * <p>
98  * A {@code ChronoZonedDateTime} is the abstract representation of an offset date-time
99  * where the {@code Chronology chronology}, or calendar system, is pluggable.
100  * The date-time is defined in terms of fields expressed by {@link TemporalField},
101  * where most common implementations are defined in {@link ChronoField}.
102  * The chronology defines how the calendar system operates and the meaning of
103  * the standard fields.
104  *
105  * <h3>When to use this interface</h3>
106  * The design of the API encourages the use of {@code ZonedDateTime} rather than this
107  * interface, even in the case where the application needs to deal with multiple
108  * calendar systems. The rationale for this is explored in detail in {@link ChronoLocalDate}.
109  * <p>
110  * Ensure that the discussion in {@code ChronoLocalDate} has been read and understood
111  * before using this interface.
112  *
113  * @implSpec
114  * This interface must be implemented with care to ensure other classes operate correctly.
115  * All implementations that can be instantiated must be final, immutable and thread-safe.
116  * Subclasses should be Serializable wherever possible.
117  *
118  * @param <D> the concrete type for the date of this date-time
119  * @since 1.8
```

```

120  */
121  public interface ChronoZonedDateTime<D extends ChronoLocalDate>
122      extends Temporal, Comparable<ChronoZonedDateTime<?>> {
123
124      /**
125       * Gets a comparator that compares {@code ChronoZonedDateTime} in
126       * time-line order ignoring the chronology.
127       * <p>
128       * This comparator differs from the comparison in {@link #compareTo} in that it
129       * only compares the underlying instant and not the chronology.
130       * This allows dates in different calendar systems to be compared based
131       * on the position of the date-time on the instant time-line.
132       * The underlying comparison is equivalent to comparing the epoch-second and nano-of-second.
133       *
134       * @return a comparator that compares in time-line order ignoring the chronology
135       * @see #isAfter
136       * @see #isBefore
137       * @see #isEqual
138       */
139      static Comparator<ChronoZonedDateTime<?>> timeLineOrder() {
140          return AbstractChronology.INSTANT_ORDER;
141      }
142
143      //-----
144      /**
145       * Obtains an instance of {@code ChronoZonedDateTime} from a temporal object.
146       * <p>
147       * This creates a zoned date-time based on the specified temporal.
148       * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
149       * which this factory converts to an instance of {@code ChronoZonedDateTime}.
150       * <p>
151       * The conversion extracts and combines the chronology, date, time and zone
152       * from the temporal object. The behavior is equivalent to using
153       * {@link Chronology#zonedDateTime(TemporalAccessor)} with the extracted chronology.
154       * Implementations are permitted to perform optimizations such as accessing
155       * those fields that are equivalent to the relevant objects.
156       * <p>
157       * This method matches the signature of the functional interface {@link TemporalQuery}
158       * allowing it to be used as a query via method reference, {@code ChronoZonedDateTime::from}.
159       *
160       * @param temporal the temporal object to convert, not null
161       * @return the date-time, not null
162       * @throws DateTimeException if unable to convert to a {@code ChronoZonedDateTime}
163       * @see Chronology#zonedDateTime(TemporalAccessor)
164       */
165      static ChronoZonedDateTime<?> from(TemporalAccessor temporal) {
166          if (temporal instanceof ChronoZonedDateTime) {
167              return (ChronoZonedDateTime<?>) temporal;
168          }
169          Objects.requireNonNull(temporal, "temporal");
170          Chronology chrono = temporal.query(TemporalQueries.chronology());
171          if (chrono == null) {
172              throw new DateTimeException("Unable to obtain ChronoZonedDateTime from TemporalAccessor

```

```

        : " + temporal.getClass());
173     }
174     return chrono.zonedDateTime(temporal);
175 }
176
177 //-----
178 @Override
179 default ValueRange range(TemporalField field) {
180     if (field instanceof ChronoField) {
181         if (field == INSTANT_SECONDS || field == OFFSET_SECONDS) {
182             return field.range();
183         }
184         return toLocalDateTime().range(field);
185     }
186     return field.rangeRefinedBy(this);
187 }
188
189 @Override
190 default int get(TemporalField field) {
191     if (field instanceof ChronoField) {
192         switch ((ChronoField) field) {
193             case INSTANT_SECONDS:
194                 throw new UnsupportedOperationException("Invalid field 'InstantSeconds' for
195                     get() method, use getLong() instead");
196             case OFFSET_SECONDS:
197                 return getOffset().getTotalSeconds();
198         }
199         return toLocalDateTime().get(field);
200     }
201     return Temporal.super.get(field);
202 }
203
204 @Override
205 default long getLong(TemporalField field) {
206     if (field instanceof ChronoField) {
207         switch ((ChronoField) field) {
208             case INSTANT_SECONDS: return toEpochSecond();
209             case OFFSET_SECONDS: return getOffset().getTotalSeconds();
210         }
211         return toLocalDateTime().getLong(field);
212     }
213     return field.getFrom(this);
214 }
215
216 /**
217  * Gets the local date part of this date-time.
218  * <p>
219  * This returns a local date with the same year, month and day
220  * as this date-time.
221  *
222  * @return the date part of this date-time, not null
223  */
224 default D toLocalDate() {

```

```
224         return toLocalDateTime().toLocalDate();
225     }
226
227     /**
228      * Gets the local time part of this date-time.
229      * <p>
230      * This returns a local time with the same hour, minute, second and
231      * nanosecond as this date-time.
232      *
233      * @return the time part of this date-time, not null
234      */
235     default LocalTime toLocalTime() {
236         return toLocalDateTime().toLocalTime();
237     }
238
239     /**
240      * Gets the local date-time part of this date-time.
241      * <p>
242      * This returns a local date with the same year, month and day
243      * as this date-time.
244      *
245      * @return the local date-time part of this date-time, not null
246      */
247     ChronoLocalDateTime<D> toLocalDateTime();
248
249     /**
250      * Gets the chronology of this date-time.
251      * <p>
252      * The {@code Chronology} represents the calendar system in use.
253      * The era and other fields in {@link ChronoField} are defined by the chronology.
254      *
255      * @return the chronology, not null
256      */
257     default Chronology getChronology() {
258         return toLocalDate().getChronology();
259     }
260
261     /**
262      * Gets the zone offset, such as '+01:00'.
263      * <p>
264      * This is the offset of the local date-time from UTC/Greenwich.
265      *
266      * @return the zone offset, not null
267      */
268     ZoneOffset getOffset();
269
270     /**
271      * Gets the zone ID, such as 'Europe/Paris'.
272      * <p>
273      * This returns the stored time-zone id used to determine the time-zone rules.
274      *
275      * @return the zone ID, not null
276      */
```

```

277     ZoneId getZone();
278
279     //-----
280     /**
281      * Returns a copy of this date-time changing the zone offset to the
282      * earlier of the two valid offsets at a local time-line overlap.
283      * <p>
284      * This method only has any effect when the local time-line overlaps, such as
285      * at an autumn daylight savings cutover. In this scenario, there are two
286      * valid offsets for the local date-time. Calling this method will return
287      * a zoned date-time with the earlier of the two selected.
288      * <p>
289      * If this method is called when it is not an overlap, {@code this}
290      * is returned.
291      * <p>
292      * This instance is immutable and unaffected by this method call.
293      *
294      * @return a {@code ChronoZonedDateTime} based on this date-time with the earlier offset, not
295      *         null
296      * @throws DateTimeException if no rules can be found for the zone
297      * @throws DateTimeException if no rules are valid for this date-time
298      */
299     ChronoZonedDateTime<D> withEarlierOffsetAtOverlap();
300
301     /**
302      * Returns a copy of this date-time changing the zone offset to the
303      * later of the two valid offsets at a local time-line overlap.
304      * <p>
305      * This method only has any effect when the local time-line overlaps, such as
306      * at an autumn daylight savings cutover. In this scenario, there are two
307      * valid offsets for the local date-time. Calling this method will return
308      * a zoned date-time with the later of the two selected.
309      * <p>
310      * If this method is called when it is not an overlap, {@code this}
311      * is returned.
312      * <p>
313      * This instance is immutable and unaffected by this method call.
314      *
315      * @return a {@code ChronoZonedDateTime} based on this date-time with the later offset, not
316      *         null
317      * @throws DateTimeException if no rules can be found for the zone
318      * @throws DateTimeException if no rules are valid for this date-time
319      */
320     ChronoZonedDateTime<D> withLaterOffsetAtOverlap();
321
322     /**
323      * Returns a copy of this date-time with a different time-zone,
324      * retaining the local date-time if possible.
325      * <p>
326      * This method changes the time-zone and retains the local date-time.
327      * The local date-time is only changed if it is invalid for the new zone.
328      * <p>
329      * To change the zone and adjust the local date-time,

```

```

328     * use {@link #withZoneSameInstant(ZoneId)}.
329     * <p>
330     * This instance is immutable and unaffected by this method call.
331     *
332     * @param zone the time-zone to change to, not null
333     * @return a {@code ChronoZonedDateTime} based on this date-time with the requested zone, not
334     *         null
335     */
336     ChronoZonedDateTime<D> withZoneSameLocal(ZoneId zone);
337
338     /**
339     * Returns a copy of this date-time with a different time-zone,
340     * retaining the instant.
341     * <p>
342     * This method changes the time-zone and retains the instant.
343     * This normally results in a change to the local date-time.
344     * <p>
345     * This method is based on retaining the same instant, thus gaps and overlaps
346     * in the local time-line have no effect on the result.
347     * <p>
348     * To change the offset while keeping the local time,
349     * use {@link #withZoneSameLocal(ZoneId)}.
350     *
351     * @param zone the time-zone to change to, not null
352     * @return a {@code ChronoZonedDateTime} based on this date-time with the requested zone, not
353     *         null
354     * @throws DateTimeException if the result exceeds the supported date range
355     */
356     ChronoZonedDateTime<D> withZoneSameInstant(ZoneId zone);
357
358     /**
359     * Checks if the specified field is supported.
360     * <p>
361     * This checks if the specified field can be queried on this date-time.
362     * If false, then calling the {@link #range(TemporalField) range},
363     * {@link #get(TemporalField) get} and {@link #with(TemporalField, long)}
364     * methods will throw an exception.
365     * <p>
366     * The set of supported fields is defined by the chronology and normally includes
367     * all {@code ChronoField} fields.
368     * <p>
369     * If the field is not a {@code ChronoField}, then the result of this method
370     * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
371     * passing {@code this} as the argument.
372     * Whether the field is supported is determined by the field.
373     *
374     * @param field the field to check, null returns false
375     * @return true if the field can be queried, false if not
376     */
377     @Override
378     boolean isSupported(TemporalField field);

```

```

379     * Checks if the specified unit is supported.
380     * <p>
381     * This checks if the specified unit can be added to or subtracted from this date-time.
382     * If false, then calling the {@link #plus(long, TemporalUnit)} and
383     * {@link #minus(long, TemporalUnit) minus} methods will throw an exception.
384     * <p>
385     * The set of supported units is defined by the chronology and normally includes
386     * all {@code ChronoUnit} units except {@code FOREVER}.
387     * <p>
388     * If the unit is not a {@code ChronoUnit}, then the result of this method
389     * is obtained by invoking {@code TemporalUnit.isSupportedBy(Temporal)}
390     * passing {@code this} as the argument.
391     * Whether the unit is supported is determined by the unit.
392     *
393     * @param unit the unit to check, null returns false
394     * @return true if the unit can be added/subtracted, false if not
395     */
396     @Override
397     default boolean isSupported(TemporalUnit unit) {
398         if (unit instanceof ChronoUnit) {
399             return unit != FOREVER;
400         }
401         return unit != null && unit.isSupportedBy(this);
402     }
403
404     //-----
405     // override for covariant return type
406     /**
407     * {@inheritDoc}
408     * @throws DateTimeException {@inheritDoc}
409     * @throws ArithmeticException {@inheritDoc}
410     */
411     @Override
412     default ChronoZonedDateTime<D> with(TemporalAdjuster adjuster) {
413         return ChronoZonedDateTimeImpl.ensureValid(getChronology(), Temporal.super.with(adjuster));
414     }
415
416     /**
417     * {@inheritDoc}
418     * @throws DateTimeException {@inheritDoc}
419     * @throws ArithmeticException {@inheritDoc}
420     */
421     @Override
422     ChronoZonedDateTime<D> with(TemporalField field, long newValue);
423
424     /**
425     * {@inheritDoc}
426     * @throws DateTimeException {@inheritDoc}
427     * @throws ArithmeticException {@inheritDoc}
428     */
429     @Override
430     default ChronoZonedDateTime<D> plus(TemporalAmount amount) {
431         return ChronoZonedDateTimeImpl.ensureValid(getChronology(), Temporal.super.plus(amount));

```

```

432     }
433
434     /**
435      * {@inheritDoc}
436      * @throws DateTimeException {@inheritDoc}
437      * @throws ArithmeticException {@inheritDoc}
438      */
439     @Override
440     ChronoZonedDateTime<D> plus(long amountToAdd, TemporalUnit unit);
441
442     /**
443      * {@inheritDoc}
444      * @throws DateTimeException {@inheritDoc}
445      * @throws ArithmeticException {@inheritDoc}
446      */
447     @Override
448     default ChronoZonedDateTime<D> minus(TemporalAmount amount) {
449         return ChronoZonedDateTimeImpl.ensureValid(getChronology(), Temporal.super.minus(amount));
450     }
451
452     /**
453      * {@inheritDoc}
454      * @throws DateTimeException {@inheritDoc}
455      * @throws ArithmeticException {@inheritDoc}
456      */
457     @Override
458     default ChronoZonedDateTime<D> minus(long amountToSubtract, TemporalUnit unit) {
459         return ChronoZonedDateTimeImpl.ensureValid(getChronology(), Temporal.super.minus(
460             amountToSubtract, unit));
461     }
462
463     //-----
464     /**
465      * Queries this date-time using the specified query.
466      * <p>
467      * This queries this date-time using the specified query strategy object.
468      * The {@code TemporalQuery} object defines the logic to be used to
469      * obtain the result. Read the documentation of the query to understand
470      * what the result of this method will be.
471      * <p>
472      * The result of this method is obtained by invoking the
473      * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
474      * specified query passing {@code this} as the argument.
475      *
476      * @param <R> the type of the result
477      * @param query the query to invoke, not null
478      * @return the query result, null may be returned (defined by the query)
479      * @throws DateTimeException if unable to query (defined by the query)
480      * @throws ArithmeticException if numeric overflow occurs (defined by the query)
481      */
482     @SuppressWarnings("unchecked")
483     @Override
484     default <R> R query(TemporalQuery<R> query) {

```



```

484     if (query == TemporalQueries.zone() || query == TemporalQueries.zoneId()) {
485         return (R) getZone();
486     } else if (query == TemporalQueries.offset()) {
487         return (R) getOffset();
488     } else if (query == TemporalQueries.localTime()) {
489         return (R) toLocalTime();
490     } else if (query == TemporalQueries.chronology()) {
491         return (R) getChronology();
492     } else if (query == TemporalQueries.precision()) {
493         return (R) NANOS;
494     }
495     // inline TemporalAccessor.super.query(query) as an optimization
496     // non-JDK classes are not permitted to make this optimization
497     return query.queryFrom(this);
498 }
499
500 /**
501  * Formats this date-time using the specified formatter.
502  * <p>
503  * This date-time will be passed to the formatter to produce a string.
504  * <p>
505  * The default implementation must behave as follows:
506  * <pre>
507  * return formatter.format(this);
508  * </pre>
509  *
510  * @param formatter the formatter to use, not null
511  * @return the formatted date-time string, not null
512  * @throws DateTimeException if an error occurs during printing
513  */
514 default String format(DateTimeFormatter formatter) {
515     Objects.requireNonNull(formatter, "formatter");
516     return formatter.format(this);
517 }
518
519 //-----
520 /**
521  * Converts this date-time to an {@code Instant}.
522  * <p>
523  * This returns an {@code Instant} representing the same point on the
524  * time-line as this date-time. The calculation combines the
525  * {@linkplain #toLocalDateTime() local date-time} and
526  * {@linkplain #getOffset() offset}.
527  *
528  * @return an {@code Instant} representing the same instant, not null
529  */
530 default Instant toInstant() {
531     return Instant.ofEpochSecond(toEpochSecond(), toLocalTime().getNano());
532 }
533
534 /**
535  * Converts this date-time to the number of seconds from the epoch
536  * of 1970-01-01T00:00:00Z.

```

```

537     * <p>
538     * This uses the {@linkplain #toLocalDateTime() local date-time} and
539     * {@linkplain #getOffset() offset} to calculate the epoch-second value,
540     * which is the number of elapsed seconds from 1970-01-01T00:00:00Z.
541     * Instants on the time-line after the epoch are positive, earlier are negative.
542     *
543     * @return the number of seconds from the epoch of 1970-01-01T00:00:00Z
544     */
545     default long toEpochSecond() {
546         long epochDay = toLocalDate().toEpochDay();
547         long secs = epochDay * 86400 + toLocalTime().toSecondOfDay();
548         secs -= getOffset().getTotalSeconds();
549         return secs;
550     }
551
552     //-----
553     /**
554     * Compares this date-time to another date-time, including the chronology.
555     * <p>
556     * The comparison is based first on the instant, then on the local date-time,
557     * then on the zone ID, then on the chronology.
558     * It is "consistent with equals", as defined by {@link Comparable}.
559     * <p>
560     * If all the date-time objects being compared are in the same chronology, then the
561     * additional chronology stage is not required.
562     * <p>
563     * This default implementation performs the comparison defined above.
564     *
565     * @param other the other date-time to compare to, not null
566     * @return the comparator value, negative if less, positive if greater
567     */
568     @Override
569     default int compareTo(ChronoZonedDateTime<?> other) {
570         int cmp = Long.compare(toEpochSecond(), other.toEpochSecond());
571         if (cmp == 0) {
572             cmp = toLocalTime().getNano() - other.toLocalTime().getNano();
573             if (cmp == 0) {
574                 cmp = toLocalDateTime().compareTo(other.toLocalDateTime());
575                 if (cmp == 0) {
576                     cmp = getZone().getId().compareTo(other.getZone().getId());
577                     if (cmp == 0) {
578                         cmp = getChronology().compareTo(other.getChronology());
579                     }
580                 }
581             }
582         }
583         return cmp;
584     }
585
586     /**
587     * Checks if the instant of this date-time is before that of the specified date-time.
588     * <p>
589     * This method differs from the comparison in {@link #compareTo} in that it

```

```

590     * only compares the instant of the date-time. This is equivalent to using
591     * {@code dateTime1.toInstant().isBefore(dateTime2.toInstant());}.
592     * <p>
593     * This default implementation performs the comparison based on the epoch-second
594     * and nano-of-second.
595     *
596     * @param other the other date-time to compare to, not null
597     * @return true if this point is before the specified date-time
598     */
599     default boolean isBefore(ChronoZonedDateTime<?> other) {
600         long thisEpochSec = toEpochSecond();
601         long otherEpochSec = other.toEpochSecond();
602         return thisEpochSec < otherEpochSec ||
603             (thisEpochSec == otherEpochSec && toLocalTime().getNano() < other.toLocalTime().getNano());
604     }
605
606     /**
607     * Checks if the instant of this date-time is after that of the specified date-time.
608     * <p>
609     * This method differs from the comparison in {@link #compareTo} in that it
610     * only compares the instant of the date-time. This is equivalent to using
611     * {@code dateTime1.toInstant().isAfter(dateTime2.toInstant());}.
612     * <p>
613     * This default implementation performs the comparison based on the epoch-second
614     * and nano-of-second.
615     *
616     * @param other the other date-time to compare to, not null
617     * @return true if this is after the specified date-time
618     */
619     default boolean isAfter(ChronoZonedDateTime<?> other) {
620         long thisEpochSec = toEpochSecond();
621         long otherEpochSec = other.toEpochSecond();
622         return thisEpochSec > otherEpochSec ||
623             (thisEpochSec == otherEpochSec && toLocalTime().getNano() > other.toLocalTime().getNano());
624     }
625
626     /**
627     * Checks if the instant of this date-time is equal to that of the specified date-time.
628     * <p>
629     * This method differs from the comparison in {@link #compareTo} and {@link #equals}
630     * in that it only compares the instant of the date-time. This is equivalent to using
631     * {@code dateTime1.toInstant().equals(dateTime2.toInstant());}.
632     * <p>
633     * This default implementation performs the comparison based on the epoch-second
634     * and nano-of-second.
635     *
636     * @param other the other date-time to compare to, not null
637     * @return true if the instant equals the instant of the specified date-time
638     */
639     default boolean isEqual(ChronoZonedDateTime<?> other) {
640         return toEpochSecond() == other.toEpochSecond() &&

```

```

641         toLocalTime().getNano() == other.toLocalTime().getNano();
642     }
643
644     //-----
645     /**
646      * Checks if this date-time is equal to another date-time.
647      * <p>
648      * The comparison is based on the offset date-time and the zone.
649      * To compare for the same instant on the time-line, use {@link #compareTo}.
650      * Only objects of type {@code ChronoZonedDateTime} are compared, other types return false.
651      *
652      * @param obj the object to check, null returns false
653      * @return true if this is equal to the other date-time
654      */
655     @Override
656     boolean equals(Object obj);
657
658     /**
659      * A hash code for this date-time.
660      *
661      * @return a suitable hash code
662      */
663     @Override
664     int hashCode();
665
666     //-----
667     /**
668      * Outputs this date-time as a {@code String}.
669      * <p>
670      * The output will include the full zoned date-time.
671      *
672      * @return a string representation of this date-time, not null
673      */
674     @Override
675     String toString();
676
677 }

```

5.9 ChronoZonedDateTimeImpl.java

Secuencia 79: java/time/chrono/ChronoZonedDateTimeImpl.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *

```

```
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.chrono;
63
64  import static java.time.temporal.ChronoUnit.SECONDS;
65
```

```
66 import java.io.IOException;
67 import java.io.InvalidObjectException;
68 import java.io.ObjectInput;
69 import java.io.ObjectInputStream;
70 import java.io.ObjectOutput;
71 import java.io.Serializable;
72 import java.time.Instant;
73 import java.time.LocalDateTime;
74 import java.time.ZoneId;
75 import java.time.ZoneOffset;
76 import java.time.temporal.ChronoField;
77 import java.time.temporal.ChronoUnit;
78 import java.time.temporal.Temporal;
79 import java.time.temporal.TemporalField;
80 import java.time.temporal.TemporalUnit;
81 import java.time.zone.ZoneOffsetTransition;
82 import java.time.zone.ZoneRules;
83 import java.util.List;
84 import java.util.Objects;
85
86 /**
87  * A date-time with a time-zone in the calendar neutral API.
88  * <p>
89  * {@code ZoneChronoDateTime} is an immutable representation of a date-time with a time-zone.
90  * This class stores all date and time fields, to a precision of nanoseconds,
91  * as well as a time-zone and zone offset.
92  * <p>
93  * The purpose of storing the time-zone is to distinguish the ambiguous case where
94  * the local time-line overlaps, typically as a result of the end of daylight time.
95  * Information about the local-time can be obtained using methods on the time-zone.
96  *
97  * @implSpec
98  * This class is immutable and thread-safe.
99  *
100  * @serial Document the delegation of this class in the serialized-form specification.
101  * @param <D> the concrete type for the date of this date-time
102  * @since 1.8
103  */
104 final class ChronoZonedDateTimeImpl<D extends ChronoLocalDate>
105     implements ChronoZonedDateTime<D>, Serializable {
106
107     /**
108      * Serialization version.
109      */
110     private static final long serialVersionUID = -5261813987200935591L;
111
112     /**
113      * The local date-time.
114      */
115     private final transient ChronoLocalDateTimeImpl<D> dateTime;
116
117     /**
118      * The zone offset.
```

```

119 private final transient ZoneOffset offset;
120 /**
121  * The zone ID.
122  */
123 private final transient ZoneId zone;
124
125 //-----
126 /**
127  * Obtains an instance from a local date-time using the preferred offset if possible.
128  *
129  * @param localDateTime the local date-time, not null
130  * @param zone the zone identifier, not null
131  * @param preferredOffset the zone offset, null if no preference
132  * @return the zoned date-time, not null
133  */
134 static <R extends ChronoLocalDate> ChronoZonedDateTime<R> ofBest(
135     ChronoLocalDateTimeImpl<R> localDateTime, ZoneId zone, ZoneOffset preferredOffset) {
136     Objects.requireNonNull(localDateTime, "localDateTime");
137     Objects.requireNonNull(zone, "zone");
138     if (zone instanceof ZoneOffset) {
139         return new ChronoZonedDateTimeImpl<>(localDateTime, (ZoneOffset) zone, zone);
140     }
141     ZoneRules rules = zone.getRules();
142     LocalDateTime isoLDT = LocalDateTime.from(localDateTime);
143     List<ZoneOffset> validOffsets = rules.getValidOffsets(isoLDT);
144     ZoneOffset offset;
145     if (validOffsets.size() == 1) {
146         offset = validOffsets.get(0);
147     } else if (validOffsets.size() == 0) {
148         ZoneOffsetTransition trans = rules.getTransition(isoLDT);
149         localDateTime = localDateTime.plusSeconds(trans.getDuration().getSeconds());
150         offset = trans.getOffsetAfter();
151     } else {
152         if (preferredOffset != null && validOffsets.contains(preferredOffset)) {
153             offset = preferredOffset;
154         } else {
155             offset = validOffsets.get(0);
156         }
157     }
158     Objects.requireNonNull(offset, "offset"); // protect against bad ZoneRules
159     return new ChronoZonedDateTimeImpl<>(localDateTime, offset, zone);
160 }
161
162 /**
163  * Obtains an instance from an instant using the specified time-zone.
164  *
165  * @param chrono the chronology, not null
166  * @param instant the instant, not null
167  * @param zone the zone identifier, not null
168  * @return the zoned date-time, not null
169  */
170 static ChronoZonedDateTimeImpl<?> ofInstant(Chronology chrono, Instant instant, ZoneId zone) {
171     ZoneRules rules = zone.getRules();

```

```

172     ZoneOffset offset = rules.getOffset(instant);
173     Objects.requireNonNull(offset, "offset"); // protect against bad ZoneRules
174     LocalDateTime ldt = LocalDateTime.ofEpochSecond(instant.getEpochSecond(), instant.getNano()
175         , offset);
176     ChronoLocalDateTimeImpl<?> cldt = (ChronoLocalDateTimeImpl<?>)chrono.localDateTime(ldt);
177     return new ChronoZonedDateTimeImpl<>(cldt, offset, zone);
178 }
179 /**
180  * Obtains an instance from an {@code Instant}.
181  *
182  * @param instant the instant to create the date-time from, not null
183  * @param zone the time-zone to use, validated not null
184  * @return the zoned date-time, validated not null
185  */
186 @SuppressWarnings("unchecked")
187 private ChronoZonedDateTimeImpl<D> create(Instant instant, ZoneId zone) {
188     return (ChronoZonedDateTimeImpl<D>)ofInstant(getChronology(), instant, zone);
189 }
190 /**
191  * Casts the {@code Temporal} to {@code ChronoZonedDateTimeImpl} ensuring it has the specified
192  * chronology.
193  *
194  * @param chrono the chronology to check for, not null
195  * @param temporal a date-time to cast, not null
196  * @return the date-time checked and cast to {@code ChronoZonedDateTimeImpl}, not null
197  * @throws ClassCastException if the date-time cannot be cast to ChronoZonedDateTimeImpl
198  * or the chronology is not equal this Chronology
199  */
200 static <R extends ChronoLocalDate> ChronoZonedDateTimeImpl<R> ensureValid(Chronology chrono,
201     Temporal temporal) {
202     @SuppressWarnings("unchecked")
203     ChronoZonedDateTimeImpl<R> other = (ChronoZonedDateTimeImpl<R>) temporal;
204     if (chrono.equals(other.getChronology()) == false) {
205         throw new ClassCastException("Chronology mismatch, required: " + chrono.getId()
206             + ", actual: " + other.getChronology().getId());
207     }
208     return other;
209 }
210 //-----
211 /**
212  * Constructor.
213  *
214  * @param dateTime the date-time, not null
215  * @param offset the zone offset, not null
216  * @param zone the zone ID, not null
217  */
218 private ChronoZonedDateTimeImpl(ChronoLocalDateTimeImpl<D> dateTime, ZoneOffset offset, ZoneId
219     zone) {
220     this.dateTime = Objects.requireNonNull(dateTime, "dateTime");
221     this.offset = Objects.requireNonNull(offset, "offset");

```



```

221     this.zone = Objects.requireNonNull(zone, "zone");
222 }
223
224 //-----
225 @Override
226 public ZoneOffset getOffset() {
227     return offset;
228 }
229
230 @Override
231 public ChronoZonedDateTime<D> withEarlierOffsetAtOverlap() {
232     ZoneOffsetTransition trans = getZone().getRules().getTransition(LocalDateTime.from(this));
233     if (trans != null && trans.isOverlap()) {
234         ZoneOffset earlierOffset = trans.getOffsetBefore();
235         if (earlierOffset.equals(offset) == false) {
236             return new ChronoZonedDateTimeImpl<>(dateTime, earlierOffset, zone);
237         }
238     }
239     return this;
240 }
241
242 @Override
243 public ChronoZonedDateTime<D> withLaterOffsetAtOverlap() {
244     ZoneOffsetTransition trans = getZone().getRules().getTransition(LocalDateTime.from(this));
245     if (trans != null) {
246         ZoneOffset offset = trans.getOffsetAfter();
247         if (offset.equals(getOffset()) == false) {
248             return new ChronoZonedDateTimeImpl<>(dateTime, offset, zone);
249         }
250     }
251     return this;
252 }
253
254 //-----
255 @Override
256 public ChronoLocalDate<D> toLocalDate() {
257     return dateTime;
258 }
259
260 @Override
261 public ZoneId getZone() {
262     return zone;
263 }
264
265 @Override
266 public ChronoZonedDateTime<D> withZoneSameLocal(ZoneId zone) {
267     return ofBest(dateTime, zone, offset);
268 }
269
270 @Override
271 public ChronoZonedDateTime<D> withZoneSameInstant(ZoneId zone) {
272     Objects.requireNonNull(zone, "zone");
273     return this.zone.equals(zone) ? this : create(dateTime.toInstant(offset), zone);

```

```

274     }
275
276     //-----
277     @Override
278     public boolean isSupported(TemporalField field) {
279         return field instanceof ChronoField || (field != null && field.isSupportedBy(this));
280     }
281
282     //-----
283     @Override
284     public ChronoZonedDateTime<D> with(TemporalField field, long newValue) {
285         if (field instanceof ChronoField) {
286             ChronoField f = (ChronoField) field;
287             switch (f) {
288                 case INSTANT_SECONDS: return plus(newValue - toEpochSecond(), SECONDS);
289                 case OFFSET_SECONDS: {
290                     ZoneOffset offset = ZoneOffset.ofTotalSeconds(f.checkValidIntValue(newValue));
291                     return create(dateTime.toInstant(offset), zone);
292                 }
293             }
294             return ofBest(dateTime.with(field, newValue), zone, offset);
295         }
296         return ChronoZonedDateTimeImpl.ensureValid(getChronology(), field.adjustInto(this, newValue));
297     }
298
299     //-----
300     @Override
301     public ChronoZonedDateTime<D> plus(long amountToAdd, TemporalUnit unit) {
302         if (unit instanceof ChronoUnit) {
303             return with(dateTime.plus(amountToAdd, unit));
304         }
305         return ChronoZonedDateTimeImpl.ensureValid(getChronology(), unit.addTo(this, amountToAdd));
306         // TODO: Generics replacement Risk!
307     }
308
309     //-----
310     @Override
311     public long until(Temporal endExclusive, TemporalUnit unit) {
312         Objects.requireNonNull(endExclusive, "endExclusive");
313         @SuppressWarnings("unchecked")
314         ChronoZonedDateTime<D> end = (ChronoZonedDateTime<D>) getChronology().zonedDateTime(
315             endExclusive);
316         if (unit instanceof ChronoUnit) {
317             end = end.withZoneSameInstant(offset);
318             return dateTime.until(end.toLocalDateTime(), unit);
319         }
320         Objects.requireNonNull(unit, "unit");
321         return unit.between(this, end);
322     }
323     //-----
324     /**

```

```

324     * Writes the ChronoZonedDateTime using a
325     * <a href="../../../serialized-form.html#java.time.chrono.Ser">dedicated serialized form</a>.
326     * @serialData
327     * <pre>
328     *   out.writeByte(3);           // identifies a ChronoZonedDateTime
329     *   out.writeObject(toLocalDateTime());
330     *   out.writeObject(getOffset());
331     *   out.writeObject(getZone());
332     * </pre>
333     *
334     * @return the instance of {@code Ser}, not null
335     */
336     private Object writeReplace() {
337         return new Ser(Ser.CHRONO_ZONE_DATE_TIME_TYPE, this);
338     }
339
340     /**
341     * Defend against malicious streams.
342     *
343     * @param s the stream to read
344     * @throws InvalidObjectException always
345     */
346     private void readObject(ObjectInputStream s) throws InvalidObjectException {
347         throw new InvalidObjectException("Deserialization via serialization delegate");
348     }
349
350     void writeExternal(ObjectOutput out) throws IOException {
351         out.writeObject(dateTime);
352         out.writeObject(offset);
353         out.writeObject(zone);
354     }
355
356     static ChronoZonedDateTime<?> readExternal(ObjectInput in) throws IOException,
        ClassNotFoundException {
357         ChronoLocalDateTime<?> dateTime = (ChronoLocalDateTime<?>) in.readObject();
358         ZoneOffset offset = (ZoneOffset) in.readObject();
359         ZoneId zone = (ZoneId) in.readObject();
360         return dateTime.atZone(offset).withZoneSameLocal(zone);
361         // TODO: ZDT uses ofLenient()
362     }
363
364     //-----
365     @Override
366     public boolean equals(Object obj) {
367         if (this == obj) {
368             return true;
369         }
370         if (obj instanceof ChronoZonedDateTime) {
371             return compareTo((ChronoZonedDateTime<?>) obj) == 0;
372         }
373         return false;
374     }
375

```

```

376     @Override
377     public int hashCode() {
378         return toLocalDateTime().hashCode() ^ getOffset().hashCode() ^ Integer.rotateLeft(getZone()
379             .hashCode(), 3);
380     }
381     @Override
382     public String toString() {
383         String str = toLocalDateTime().toString() + getOffset().toString();
384         if (getOffset() != getZone()) {
385             str += '[' + getZone().toString() + ']';
386         }
387         return str;
388     }
389
390
391 }

```

5.10 Chronology.java

Secuencia 80: java/time/chrono/Chronology.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos

```

```
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.chrono;
63
64  import java.time.Clock;
65  import java.time.DateTimeException;
66  import java.time.Instant;
67  import java.time.LocalDate;
68  import java.time.LocalDateTime;
69  import java.time.ZoneId;
70  import java.time.format.DateTimeFormatterBuilder;
71  import java.time.format.ResolverStyle;
72  import java.time.format.TextStyle;
73  import java.time.temporal.ChronoField;
74  import java.time.temporal.TemporalAccessor;
75  import java.time.temporal.TemporalField;
76  import java.time.temporal.TemporalQueries;
77  import java.time.temporal.TemporalQuery;
78  import java.time.temporal.UnsupportedTemporalTypeException;
79  import java.time.temporal.ValueRange;
80  import java.util.List;
81  import java.util.Locale;
82  import java.util.Map;
83  import java.util.Objects;
84  import java.util.Set;
85
```

```

86 /**
87  * A calendar system, used to organize and identify dates.
88  * <p>
89  * The main date and time API is built on the ISO calendar system.
90  * The chronology operates behind the scenes to represent the general concept of a calendar system.
91  * For example, the Japanese, Minguo, Thai Buddhist and others.
92  * <p>
93  * Most other calendar systems also operate on the shared concepts of year, month and day,
94  * linked to the cycles of the Earth around the Sun, and the Moon around the Earth.
95  * These shared concepts are defined by {@link ChronoField} and are available
96  * for use by any {@code Chronology} implementation:
97  * <pre>
98  *     LocalDate isoDate = ...
99  *     ThaiBuddhistDate thaiDate = ...
100  *     int isoYear = isoDate.get(ChronoField.YEAR);
101  *     int thaiYear = thaiDate.get(ChronoField.YEAR);
102  * </pre>
103  * As shown, although the date objects are in different calendar systems, represented by different
104  * {@code Chronology} instances, both can be queried using the same constant on {@code ChronoField}
105  *   .
106  * For a full discussion of the implications of this, see {@link ChronoLocalDate}.
107  * In general, the advice is to use the known ISO-based {@code LocalDate}, rather than
108  *   {@code ChronoLocalDate}.
109  * <p>
110  * While a {@code Chronology} object typically uses {@code ChronoField} and is based on
111  * an era, year-of-era, month-of-year, day-of-month model of a date, this is not required.
112  * A {@code Chronology} instance may represent a totally different kind of calendar system,
113  * such as the Mayan.
114  * <p>
115  * In practical terms, the {@code Chronology} instance also acts as a factory.
116  * The {@link #of(String)} method allows an instance to be looked up by identifier,
117  * while the {@link #of(Locale(Locale))} method allows lookup by locale.
118  * <p>
119  * The {@code Chronology} instance provides a set of methods to create {@code ChronoLocalDate}
120  *   instances.
121  * The date classes are used to manipulate specific dates.
122  * <ul>
123  * <li> {@link #dateNow() dateNow()}
124  * <li> {@link #dateNow(Clock) dateNow(clock)}
125  * <li> {@link #dateNow(ZoneId) dateNow(zone)}
126  * <li> {@link #date(int, int, int) date(yearProleptic, month, day)}
127  * <li> {@link #date(Era, int, int, int) date(era, yearOfEra, month, day)}
128  * <li> {@link #dateYearDay(int, int) dateYearDay(yearProleptic, dayOfYear)}
129  * <li> {@link #dateYearDay(Era, int, int) dateYearDay(era, yearOfEra, dayOfYear)}
130  * <li> {@link #date(TemporalAccessor) date(TemporalAccessor)}
131  * </ul>
132  *
133  * <h3 id="addcalendars">Adding New Calendars</h3>
134  * The set of available chronologies can be extended by applications.
135  * Adding a new calendar system requires the writing of an implementation of
136  *   {@code Chronology}, {@code ChronoLocalDate} and {@code Era}.
137  * The majority of the logic specific to the calendar system will be in the
138  *   {@code ChronoLocalDate} implementation.

```

```

137 * The {@code Chronology} implementation acts as a factory.
138 * <p>
139 * To permit the discovery of additional chronologies, the {@link java.util.ServiceLoader
    ServiceLoader}
140 * is used. A file must be added to the {@code META-INF/services} directory with the
141 * name 'java.time.chrono.Chronology' listing the implementation classes.
142 * See the ServiceLoader for more details on service loading.
143 * For lookup by id or calendarType, the system provided calendars are found
144 * first followed by application provided calendars.
145 * <p>
146 * Each chronology must define a chronology ID that is unique within the system.
147 * If the chronology represents a calendar system defined by the
148 * CLDR specification then the calendar type is the concatenation of the
149 * CLDR type and, if applicable, the CLDR variant,
150 *
151 * @implSpec
152 * This interface must be implemented with care to ensure other classes operate correctly.
153 * All implementations that can be instantiated must be final, immutable and thread-safe.
154 * Subclasses should be Serializable wherever possible.
155 *
156 * @since 1.8
157 */
158 public interface Chronology extends Comparable<Chronology> {
159
160     /**
161      * Obtains an instance of {@code Chronology} from a temporal object.
162      * <p>
163      * This obtains a chronology based on the specified temporal.
164      * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
165      * which this factory converts to an instance of {@code Chronology}.
166      * <p>
167      * The conversion will obtain the chronology using {@link TemporalQueries#chronology()}.
168      * If the specified temporal object does not have a chronology, {@link IsoChronology} is
        returned.
169      * <p>
170      * This method matches the signature of the functional interface {@link TemporalQuery}
171      * allowing it to be used as a query via method reference, {@code Chronology::from}.
172      *
173      * @param temporal the temporal to convert, not null
174      * @return the chronology, not null
175      * @throws DateTimeException if unable to convert to an {@code Chronology}
176      */
177     static Chronology from(TemporalAccessor temporal) {
178         Objects.requireNonNull(temporal, "temporal");
179         Chronology obj = temporal.query(TemporalQueries.chronology());
180         return (obj != null ? obj : IsoChronology.INSTANCE);
181     }
182
183     //-----
184     /**
185      * Obtains an instance of {@code Chronology} from a locale.
186      * <p>
187      * This returns a {@code Chronology} based on the specified locale,

```

```

188 * typically returning {@code IsoChronology}. Other calendar systems
189 * are only returned if they are explicitly selected within the locale.
190 * <p>
191 * The {@link Locale} class provide access to a range of information useful
192 * for localizing an application. This includes the language and region,
193 * such as "en-GB" for English as used in Great Britain.
194 * <p>
195 * The {@code Locale} class also supports an extension mechanism that
196 * can be used to identify a calendar system. The mechanism is a form
197 * of key-value pairs, where the calendar system has the key "ca".
198 * For example, the locale "en-JP-u-ca-japanese" represents the English
199 * language as used in Japan with the Japanese calendar system.
200 * <p>
201 * This method finds the desired calendar system by in a manner equivalent
202 * to passing "ca" to {@link Locale#getUnicodeLocaleType(String)}.
203 * If the "ca" key is not present, then {@code IsoChronology} is returned.
204 * <p>
205 * Note that the behavior of this method differs from the older
206 * {@link java.util.Calendar#getInstance(Locale)} method.
207 * If that method receives a locale of "th_TH" it will return {@code BuddhistCalendar}.
208 * By contrast, this method will return {@code IsoChronology}.
209 * Passing the locale "th-TH-u-ca-buddhist" into either method will
210 * result in the Thai Buddhist calendar system and is therefore the
211 * recommended approach going forward for Thai calendar system localization.
212 * <p>
213 * A similar, but simpler, situation occurs for the Japanese calendar system.
214 * The locale "jp_JP_JP" has previously been used to access the calendar.
215 * However, unlike the Thai locale, "ja_JP_JP" is automatically converted by
216 * {@code Locale} to the modern and recommended form of "ja-JP-u-ca-japanese".
217 * Thus, there is no difference in behavior between this method and
218 * {@code Calendar#getInstance(Locale)}.
219 *
220 * @param locale the locale to use to obtain the calendar system, not null
221 * @return the calendar system associated with the locale, not null
222 * @throws DateTimeException if the locale-specified calendar cannot be found
223 */
224 static Chronology ofLocale(Locale locale) {
225     return AbstractChronology.ofLocale(locale);
226 }
227
228 //-----
229 /**
230 * Obtains an instance of {@code Chronology} from a chronology ID or
231 * calendar system type.
232 * <p>
233 * This returns a chronology based on either the ID or the type.
234 * The {@link #getId() chronology ID} uniquely identifies the chronology.
235 * The {@link #getCalendarType() calendar system type} is defined by the
236 * CLDR specification.
237 * <p>
238 * The chronology may be a system chronology or a chronology
239 * provided by the application via ServiceLoader configuration.
240 * <p>

```



```

241     * Since some calendars can be customized, the ID or type typically refers
242     * to the default customization. For example, the Gregorian calendar can have multiple
243     * cutover dates from the Julian, but the lookup only provides the default cutover date.
244     *
245     * @param id the chronology ID or calendar system type, not null
246     * @return the chronology with the identifier requested, not null
247     * @throws DateTimeException if the chronology cannot be found
248     */
249     static Chronology of(String id) {
250         return AbstractChronology.of(id);
251     }
252
253     /**
254     * Returns the available chronologies.
255     * <p>
256     * Each returned {@code Chronology} is available for use in the system.
257     * The set of chronologies includes the system chronologies and
258     * any chronologies provided by the application via ServiceLoader
259     * configuration.
260     *
261     * @return the independent, modifiable set of the available chronology IDs, not null
262     */
263     static Set<Chronology> getAvailableChronologies() {
264         return AbstractChronology.getAvailableChronologies();
265     }
266
267     //-----
268     /**
269     * Gets the ID of the chronology.
270     * <p>
271     * The ID uniquely identifies the {@code Chronology}.
272     * It can be used to lookup the {@code Chronology} using {@link #of(String)}.
273     *
274     * @return the chronology ID, not null
275     * @see #getCalendarType()
276     */
277     String getId();
278
279     /**
280     * Gets the calendar type of the calendar system.
281     * <p>
282     * The calendar type is an identifier defined by the CLDR and
283     * <em>Unicode Locale Data Markup Language (LDML)</em> specifications
284     * to uniquely identification a calendar.
285     * The {@code getCalendarType} is the concatenation of the CLDR calendar type
286     * and the variant, if applicable, is appended separated by "-".
287     * The calendar type is used to lookup the {@code Chronology} using {@link #of(String)}.
288     *
289     * @return the calendar system type, null if the calendar is not defined by CLDR/LDML
290     * @see #getId()
291     */
292     String getCalendarType();
293

```

```

294 //-----
295 /**
296  * Obtains a local date in this chronology from the era, year-of-era,
297  * month-of-year and day-of-month fields.
298  *
299  * @implSpec
300  * The default implementation combines the era and year-of-era into a proleptic
301  * year before calling {@link #date(int, int, int)}.
302  *
303  * @param era the era of the correct type for the chronology, not null
304  * @param yearOfEra the chronology year-of-era
305  * @param month the chronology month-of-year
306  * @param dayOfMonth the chronology day-of-month
307  * @return the local date in this chronology, not null
308  * @throws DateTimeException if unable to create the date
309  * @throws ClassCastException if the {@code era} is not of the correct type for the chronology
310  */
311 default ChronoLocalDate date(Era era, int yearOfEra, int month, int dayOfMonth) {
312     return date(prolepticYear(era, yearOfEra), month, dayOfMonth);
313 }
314
315 /**
316  * Obtains a local date in this chronology from the proleptic-year,
317  * month-of-year and day-of-month fields.
318  *
319  * @param prolepticYear the chronology proleptic-year
320  * @param month the chronology month-of-year
321  * @param dayOfMonth the chronology day-of-month
322  * @return the local date in this chronology, not null
323  * @throws DateTimeException if unable to create the date
324  */
325 ChronoLocalDate date(int prolepticYear, int month, int dayOfMonth);
326
327 /**
328  * Obtains a local date in this chronology from the era, year-of-era and
329  * day-of-year fields.
330  *
331  * @implSpec
332  * The default implementation combines the era and year-of-era into a proleptic
333  * year before calling {@link #dateYearDay(int, int)}.
334  *
335  * @param era the era of the correct type for the chronology, not null
336  * @param yearOfEra the chronology year-of-era
337  * @param dayOfYear the chronology day-of-year
338  * @return the local date in this chronology, not null
339  * @throws DateTimeException if unable to create the date
340  * @throws ClassCastException if the {@code era} is not of the correct type for the chronology
341  */
342 default ChronoLocalDate dateYearDay(Era era, int yearOfEra, int dayOfYear) {
343     return dateYearDay(prolepticYear(era, yearOfEra), dayOfYear);
344 }
345
346 /**

```

```

347     * Obtains a local date in this chronology from the proleptic-year and
348     * day-of-year fields.
349     *
350     * @param prolepticYear the chronology proleptic-year
351     * @param dayOfYear the chronology day-of-year
352     * @return the local date in this chronology, not null
353     * @throws DateTimeException if unable to create the date
354     */
355     ChronoLocalDate dateYearDay(int prolepticYear, int dayOfYear);
356
357     /**
358     * Obtains a local date in this chronology from the epoch-day.
359     * <p>
360     * The definition of {@link ChronoField#EPOCH_DAY EPOCH_DAY} is the same
361     * for all calendar systems, thus it can be used for conversion.
362     *
363     * @param epochDay the epoch day
364     * @return the local date in this chronology, not null
365     * @throws DateTimeException if unable to create the date
366     */
367     ChronoLocalDate dateEpochDay(long epochDay);
368
369     //-----
370     /**
371     * Obtains the current local date in this chronology from the system clock in the default time-
372     * zone.
373     * <p>
374     * This will query the {@link Clock#systemDefaultZone() system clock} in the default
375     * time-zone to obtain the current date.
376     * <p>
377     * Using this method will prevent the ability to use an alternate clock for testing
378     * because the clock is hard-coded.
379     *
380     * @implSpec
381     * The default implementation invokes {@link #dateNow(Clock)}.
382     *
383     * @return the current local date using the system clock and default time-zone, not null
384     * @throws DateTimeException if unable to create the date
385     */
386     default ChronoLocalDate dateNow() {
387         return dateNow(Clock.systemDefaultZone());
388     }
389
390     /**
391     * Obtains the current local date in this chronology from the system clock in the specified
392     * time-zone.
393     * <p>
394     * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current date.
395     * Specifying the time-zone avoids dependence on the default time-zone.
396     * <p>
397     * Using this method will prevent the ability to use an alternate clock for testing
398     * because the clock is hard-coded.
399     *

```

```

398     * @implSpec
399     * The default implementation invokes {@link #dateNow(Clock)}.
400     *
401     * @param zone the zone ID to use, not null
402     * @return the current local date using the system clock, not null
403     * @throws DateTimeException if unable to create the date
404     */
405     default ChronoLocalDate dateNow(ZoneId zone) {
406         return dateNow(Clock.system(zone));
407     }
408
409     /**
410     * Obtains the current local date in this chronology from the specified clock.
411     * <p>
412     * This will query the specified clock to obtain the current date - today.
413     * Using this method allows the use of an alternate clock for testing.
414     * The alternate clock may be introduced using {@link Clock dependency injection}.
415     *
416     * @implSpec
417     * The default implementation invokes {@link #date(TemporalAccessor)}.
418     *
419     * @param clock the clock to use, not null
420     * @return the current local date, not null
421     * @throws DateTimeException if unable to create the date
422     */
423     default ChronoLocalDate dateNow(Clock clock) {
424         Objects.requireNonNull(clock, "clock");
425         return date(LocalDate.now(clock));
426     }
427
428     //-----
429     /**
430     * Obtains a local date in this chronology from another temporal object.
431     * <p>
432     * This obtains a date in this chronology based on the specified temporal.
433     * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
434     * which this factory converts to an instance of {@code ChronoLocalDate}.
435     * <p>
436     * The conversion typically uses the {@link ChronoField#EPOCH_DAY EPOCH_DAY}
437     * field, which is standardized across calendar systems.
438     * <p>
439     * This method matches the signature of the functional interface {@link TemporalQuery}
440     * allowing it to be used as a query via method reference, {@code aChronology::date}.
441     *
442     * @param temporal the temporal object to convert, not null
443     * @return the local date in this chronology, not null
444     * @throws DateTimeException if unable to create the date
445     * @see ChronoLocalDate#from(TemporalAccessor)
446     */
447     ChronoLocalDate date(TemporalAccessor temporal);
448
449     /**
450     * Obtains a local date-time in this chronology from another temporal object.

```

```

451 * <p>
452 * This obtains a date-time in this chronology based on the specified temporal.
453 * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
454 * which this factory converts to an instance of {@code ChronoLocalDateTime}.
455 * <p>
456 * The conversion extracts and combines the {@code ChronoLocalDate} and the
457 * {@code LocalTime} from the temporal object.
458 * Implementations are permitted to perform optimizations such as accessing
459 * those fields that are equivalent to the relevant objects.
460 * The result uses this chronology.
461 * <p>
462 * This method matches the signature of the functional interface {@link TemporalQuery}
463 * allowing it to be used as a query via method reference, {@code aChronology::localDateTime}.
464 *
465 * @param temporal the temporal object to convert, not null
466 * @return the local date-time in this chronology, not null
467 * @throws DateTimeException if unable to create the date-time
468 * @see ChronoLocalDateTime#from(TemporalAccessor)
469 */
470 default ChronoLocalDateTime<? extends ChronoLocalDate> localDateTime(TemporalAccessor temporal)
471 {
472     try {
473         return date(temporal).atTime(LocalTime.from(temporal));
474     } catch (DateTimeException ex) {
475         throw new DateTimeException("Unable to obtain ChronoLocalDateTime from TemporalAccessor
476             : " + temporal.getClass(), ex);
477     }
478 }
479 /**
480 * Obtains a {@code ChronoZonedDateTime} in this chronology from another temporal object.
481 * <p>
482 * This obtains a zoned date-time in this chronology based on the specified temporal.
483 * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
484 * which this factory converts to an instance of {@code ChronoZonedDateTime}.
485 * <p>
486 * The conversion will first obtain a {@code ZoneId} from the temporal object,
487 * falling back to a {@code ZoneOffset} if necessary. It will then try to obtain
488 * an {@code Instant}, falling back to a {@code ChronoLocalDateTime} if necessary.
489 * The result will be either the combination of {@code ZoneId} or {@code ZoneOffset}
490 * with {@code Instant} or {@code ChronoLocalDateTime}.
491 * Implementations are permitted to perform optimizations such as accessing
492 * those fields that are equivalent to the relevant objects.
493 * The result uses this chronology.
494 * <p>
495 * This method matches the signature of the functional interface {@link TemporalQuery}
496 * allowing it to be used as a query via method reference, {@code aChronology::zonedDateTime}.
497 *
498 * @param temporal the temporal object to convert, not null
499 * @return the zoned date-time in this chronology, not null
500 * @throws DateTimeException if unable to create the date-time
501 * @see ChronoZonedDateTime#from(TemporalAccessor)
502 */

```

```

502     default ChronoZonedDateTime<? extends ChronoLocalDate> zonedDateTime(TemporalAccessor temporal)
503     {
504         try {
505             ZoneId zone = ZoneId.from(temporal);
506             try {
507                 Instant instant = Instant.from(temporal);
508                 return zonedDateTime(instant, zone);
509             } catch (DateTimeException ex1) {
510                 ChronoLocalDateTimeImpl<?> cldt = ChronoLocalDateTimeImpl.ensureValid(this,
511                     localDateTime(temporal));
512                 return ChronoZonedDateTimeImpl.ofBest(cldt, zone, null);
513             } catch (DateTimeException ex) {
514                 throw new DateTimeException("Unable to obtain ChronoZonedDateTime from TemporalAccessor
515                     : " + temporal.getClass(), ex);
516             }
517         }
518     }
519     /**
520     * Obtains a {@code ChronoZonedDateTime} in this chronology from an {@code Instant}.
521     * <p>
522     * This obtains a zoned date-time with the same instant as that specified.
523     *
524     * @param instant the instant to create the date-time from, not null
525     * @param zone the time-zone, not null
526     * @return the zoned date-time, not null
527     * @throws DateTimeException if the result exceeds the supported range
528     */
529     default ChronoZonedDateTime<? extends ChronoLocalDate> zonedDateTime(Instant instant, ZoneId
530         zone) {
531         return ChronoZonedDateTimeImpl.ofInstant(this, instant, zone);
532     }
533     //-----
534     /**
535     * Checks if the specified year is a leap year.
536     * <p>
537     * A leap-year is a year of a longer length than normal.
538     * The exact meaning is determined by the chronology according to the following constraints.
539     * <ul>
540     * <li>a leap-year must imply a year-length longer than a non leap-year.
541     * <li>a chronology that does not support the concept of a year must return false.
542     * </ul>
543     *
544     * @param prolepticYear the proleptic-year to check, not validated for range
545     * @return true if the year is a leap year
546     */
547     boolean isLeapYear(long prolepticYear);
548     /**
549     * Calculates the proleptic-year given the era and year-of-era.
550     * <p>

```

```

551     * This combines the era and year-of-era into the single proleptic-year field.
552     * <p>
553     * If the chronology makes active use of eras, such as {@code JapaneseChronology}
554     * then the year-of-era will be validated against the era.
555     * For other chronologies, validation is optional.
556     *
557     * @param era the era of the correct type for the chronology, not null
558     * @param yearOfEra the chronology year-of-era
559     * @return the proleptic-year
560     * @throws DateTimeException if unable to convert to a proleptic-year,
561     *     such as if the year is invalid for the era
562     * @throws ClassCastException if the {@code era} is not of the correct type for the chronology
563     */
564     int prolepticYear(Era era, int yearOfEra);
565
566     /**
567     * Creates the chronology era object from the numeric value.
568     * <p>
569     * The era is, conceptually, the largest division of the time-line.
570     * Most calendar systems have a single epoch dividing the time-line into two eras.
571     * However, some have multiple eras, such as one for the reign of each leader.
572     * The exact meaning is determined by the chronology according to the following constraints.
573     * <p>
574     * The era in use at 1970-01-01 must have the value 1.
575     * Later eras must have sequentially higher values.
576     * Earlier eras must have sequentially lower values.
577     * Each chronology must refer to an enum or similar singleton to provide the era values.
578     * <p>
579     * This method returns the singleton era of the correct type for the specified era value.
580     *
581     * @param eraValue the era value
582     * @return the calendar system era, not null
583     * @throws DateTimeException if unable to create the era
584     */
585     Era eraOf(int eraValue);
586
587     /**
588     * Gets the list of eras for the chronology.
589     * <p>
590     * Most calendar systems have an era, within which the year has meaning.
591     * If the calendar system does not support the concept of eras, an empty
592     * list must be returned.
593     *
594     * @return the list of eras for the chronology, may be immutable, not null
595     */
596     List<Era> eras();
597
598     //-----
599     /**
600     * Gets the range of valid values for the specified field.
601     * <p>
602     * All fields can be expressed as a {@code long} integer.
603     * This method returns an object that describes the valid range for that value.

```

```

604     * <p>
605     * Note that the result only describes the minimum and maximum valid values
606     * and it is important not to read too much into them. For example, there
607     * could be values within the range that are invalid for the field.
608     * <p>
609     * This method will return a result whether or not the chronology supports the field.
610     *
611     * @param field the field to get the range for, not null
612     * @return the range of valid values for the field, not null
613     * @throws DateTimeException if the range for the field cannot be obtained
614     */
615     ValueRange range(ChronoField field);
616
617     //-----
618     /**
619     * Gets the textual representation of this chronology.
620     * <p>
621     * This returns the textual name used to identify the chronology,
622     * suitable for presentation to the user.
623     * The parameters control the style of the returned text and the locale.
624     *
625     * @implSpec
626     * The default implementation behaves as though the formatter was used to
627     * format the chronology textual name.
628     *
629     * @param style the style of the text required, not null
630     * @param locale the locale to use, not null
631     * @return the text value of the chronology, not null
632     */
633     default String getDisplayName(TextStyle style, Locale locale) {
634         TemporalAccessor temporal = new TemporalAccessor() {
635             @Override
636             public boolean isSupported(TemporalField field) {
637                 return false;
638             }
639             @Override
640             public long getLong(TemporalField field) {
641                 throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
642             }
643             @SuppressWarnings("unchecked")
644             @Override
645             public <R> R query(TemporalQuery<R> query) {
646                 if (query == TemporalQueries.chronology()) {
647                     return (R) Chronology.this;
648                 }
649                 return TemporalAccessor.super.query(query);
650             }
651         };
652         return new DateTimeFormatterBuilder().appendChronologyText(style).toFormatter(locale).
653             format(temporal);
654     }
655     //-----

```



```

656 /**
657  * Resolves parsed {@code ChronoField} values into a date during parsing.
658  * <p>
659  * Most {@code TemporalField} implementations are resolved using the
660  * resolve method on the field. By contrast, the {@code ChronoField} class
661  * defines fields that only have meaning relative to the chronology.
662  * As such, {@code ChronoField} date fields are resolved here in the
663  * context of a specific chronology.
664  * <p>
665  * The default implementation, which explains typical resolve behaviour,
666  * is provided in {@link AbstractChronology}.
667  *
668  * @param fieldValues the map of fields to values, which can be updated, not null
669  * @param resolverStyle the requested type of resolve, not null
670  * @return the resolved date, null if insufficient information to create a date
671  * @throws DateTimeException if the date cannot be resolved, typically
672  *         because of a conflict in the input data
673  */
674 ChronoLocalDate resolveDate(Map<TemporalField, Long> fieldValues, ResolverStyle resolverStyle);
675
676 //-----
677 /**
678  * Obtains a period for this chronology based on years, months and days.
679  * <p>
680  * This returns a period tied to this chronology using the specified
681  * years, months and days. All supplied chronologies use periods
682  * based on years, months and days, however the {@code ChronoPeriod} API
683  * allows the period to be represented using other units.
684  *
685  * @implSpec
686  * The default implementation returns an implementation class suitable
687  * for most calendar systems. It is based solely on the three units.
688  * Normalization, addition and subtraction derive the number of months
689  * in a year from the {@link #range(ChronoField)}. If the number of
690  * months within a year is fixed, then the calculation approach for
691  * addition, subtraction and normalization is slightly different.
692  * <p>
693  * If implementing an unusual calendar system that is not based on
694  * years, months and days, or where you want direct control, then
695  * the {@code ChronoPeriod} interface must be directly implemented.
696  * <p>
697  * The returned period is immutable and thread-safe.
698  *
699  * @param years the number of years, may be negative
700  * @param months the number of years, may be negative
701  * @param days the number of years, may be negative
702  * @return the period in terms of this chronology, not null
703  */
704 default ChronoPeriod period(int years, int months, int days) {
705     return new ChronoPeriodImpl(this, years, months, days);
706 }
707
708 //-----

```

```

709  /**
710   * Compares this chronology to another chronology.
711   * <p>
712   * The comparison order first by the chronology ID string, then by any
713   * additional information specific to the subclass.
714   * It is "consistent with equals", as defined by {@link Comparable}.
715   *
716   * @param other the other chronology to compare to, not null
717   * @return the comparator value, negative if less, positive if greater
718   */
719  @Override
720  int compareTo(Chronology other);
721
722  /**
723   * Checks if this chronology is equal to another chronology.
724   * <p>
725   * The comparison is based on the entire state of the object.
726   *
727   * @param obj the object to check, null returns false
728   * @return true if this is equal to the other chronology
729   */
730  @Override
731  boolean equals(Object obj);
732
733  /**
734   * A hash code for this chronology.
735   * <p>
736   * The hash code should be based on the entire state of the object.
737   *
738   * @return a suitable hash code
739   */
740  @Override
741  int hashCode();
742
743  //-----
744  /**
745   * Outputs this chronology as a {@code String}.
746   * <p>
747   * The format should include the entire state of the object.
748   *
749   * @return a string representation of this chronology, not null
750   */
751  @Override
752  String toString();
753
754 }

```

5.11 Era.java

Secuencia 81: java/time/chrono/Era.java

```

1  /*
2   * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3   * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.

```

```
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
```

```

57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.chrono;
63
64  import static java.time.temporal.ChronoField.ERA;
65  import static java.time.temporal.ChronoUnit.ERAS;
66
67  import java.time.DateTimeException;
68  import java.time.temporal.UnsupportedTemporalTypeException;
69  import java.time.format.DateTimeFormatterBuilder;
70  import java.time.format.TextStyle;
71  import java.time.temporal.ChronoField;
72  import java.time.temporal.Temporal;
73  import java.time.temporal.TemporalAccessor;
74  import java.time.temporal.TemporalAdjuster;
75  import java.time.temporal.TemporalField;
76  import java.time.temporal.TemporalQueries;
77  import java.time.temporal.TemporalQuery;
78  import java.time.temporal.ValueRange;
79  import java.util.Locale;
80
81  /**
82   * An era of the time-line.
83   * <p>
84   * Most calendar systems have a single epoch dividing the time-line into two eras.
85   * However, some calendar systems, have multiple eras, such as one for the reign
86   * of each leader.
87   * In all cases, the era is conceptually the largest division of the time-line.
88   * Each chronology defines the Era's that are known Eras and a
89   * {@link Chronology#eras Chronology.eras} to get the valid eras.
90   * <p>
91   * For example, the Thai Buddhist calendar system divides time into two eras,
92   * before and after a single date. By contrast, the Japanese calendar system
93   * has one era for the reign of each Emperor.
94   * <p>
95   * Instances of {@code Era} may be compared using the {@code ==} operator.
96   *
97   * @implSpec
98   * This interface must be implemented with care to ensure other classes operate correctly.
99   * All implementations must be singletons - final, immutable and thread-safe.
100  * It is recommended to use an enum whenever possible.
101  *
102  * @since 1.8
103  */
104  public interface Era extends TemporalAccessor, TemporalAdjuster {
105
106      /**
107       * Gets the numeric value associated with the era as defined by the chronology.
108       * Each chronology defines the predefined Eras and methods to list the Eras
109       * of the chronology.

```

```

110     * <p>
111     * All fields, including eras, have an associated numeric value.
112     * The meaning of the numeric value for era is determined by the chronology
113     * according to these principles:
114     * <ul>
115     * <li>The era in use at the epoch 1970-01-01 (ISO) has the value 1.
116     * <li>Later eras have sequentially higher values.
117     * <li>Earlier eras have sequentially lower values, which may be negative.
118     * </ul>
119     *
120     * @return the numeric era value
121     */
122     int getValue();
123
124     //-----
125     /**
126     * Checks if the specified field is supported.
127     * <p>
128     * This checks if this era can be queried for the specified field.
129     * If false, then calling the {@link #range(TemporalField) range} and
130     * {@link #get(TemporalField) get} methods will throw an exception.
131     * <p>
132     * If the field is a {@link ChronoField} then the query is implemented here.
133     * The {@code ERA} field returns true.
134     * All other {@code ChronoField} instances will return false.
135     * <p>
136     * If the field is not a {@code ChronoField}, then the result of this method
137     * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
138     * passing {@code this} as the argument.
139     * Whether the field is supported is determined by the field.
140     *
141     * @param field the field to check, null returns false
142     * @return true if the field is supported on this era, false if not
143     */
144     @Override
145     default boolean isSupported(TemporalField field) {
146         if (field instanceof ChronoField) {
147             return field == ERA;
148         }
149         return field != null && field.isSupportedBy(this);
150     }
151
152     /**
153     * Gets the range of valid values for the specified field.
154     * <p>
155     * The range object expresses the minimum and maximum valid values for a field.
156     * This era is used to enhance the accuracy of the returned range.
157     * If it is not possible to return the range, because the field is not supported
158     * or for some other reason, an exception is thrown.
159     * <p>
160     * If the field is a {@link ChronoField} then the query is implemented here.
161     * The {@code ERA} field returns the range.
162     * All other {@code ChronoField} instances will throw an {@code

```

```

        UnsupportedTemporalTypeException}.
163     * <p>
164     * If the field is not a {@code ChronoField}, then the result of this method
165     * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
166     * passing {@code this} as the argument.
167     * Whether the range can be obtained is determined by the field.
168     * <p>
169     * The default implementation must return a range for {@code ERA} from
170     * zero to one, suitable for two era calendar systems such as ISO.
171     *
172     * @param field the field to query the range for, not null
173     * @return the range of valid values for the field, not null
174     * @throws DateTimeException if the range for the field cannot be obtained
175     * @throws UnsupportedTemporalTypeException if the unit is not supported
176     */
177     @Override // override for Javadoc
178     default ValueRange range(TemporalField field) {
179         return TemporalAccessor.super.range(field);
180     }
181
182     /**
183     * Gets the value of the specified field from this era as an {@code int}.
184     * <p>
185     * This queries this era for the value of the specified field.
186     * The returned value will always be within the valid range of values for the field.
187     * If it is not possible to return the value, because the field is not supported
188     * or for some other reason, an exception is thrown.
189     * <p>
190     * If the field is a {@link ChronoField} then the query is implemented here.
191     * The {@code ERA} field returns the value of the era.
192     * All other {@code ChronoField} instances will throw an {@code
193         UnsupportedTemporalTypeException}.
194     * <p>
195     * If the field is not a {@code ChronoField}, then the result of this method
196     * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
197     * passing {@code this} as the argument. Whether the value can be obtained,
198     * and what the value represents, is determined by the field.
199     *
200     * @param field the field to get, not null
201     * @return the value for the field
202     * @throws DateTimeException if a value for the field cannot be obtained or
203     *         the value is outside the range of valid values for the field
204     * @throws UnsupportedTemporalTypeException if the field is not supported or
205     *         the range of values exceeds an {@code int}
206     * @throws ArithmeticException if numeric overflow occurs
207     */
208     @Override // override for Javadoc and performance
209     default int get(TemporalField field) {
210         if (field == ERA) {
211             return getValue();
212         }
213         return TemporalAccessor.super.get(field);
214     }

```

```

214
215 /**
216  * Gets the value of the specified field from this era as a {@code long}.
217  * <p>
218  * This queries this era for the value of the specified field.
219  * If it is not possible to return the value, because the field is not supported
220  * or for some other reason, an exception is thrown.
221  * <p>
222  * If the field is a {@link ChronoField} then the query is implemented here.
223  * The {@code ERA} field returns the value of the era.
224  * All other {@code ChronoField} instances will throw an {@code
    UnsupportedTemporalTypeException}.
225  * <p>
226  * If the field is not a {@code ChronoField}, then the result of this method
227  * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
228  * passing {@code this} as the argument. Whether the value can be obtained,
229  * and what the value represents, is determined by the field.
230  *
231  * @param field the field to get, not null
232  * @return the value for the field
233  * @throws DateTimeException if a value for the field cannot be obtained
234  * @throws UnsupportedTemporalTypeException if the field is not supported
235  * @throws ArithmeticException if numeric overflow occurs
236  */
237 @Override
238 default long getLong(TemporalField field) {
239     if (field == ERA) {
240         return getValue();
241     } else if (field instanceof ChronoField) {
242         throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
243     }
244     return field.getFrom(this);
245 }
246
247 //-----
248 /**
249  * Queries this era using the specified query.
250  * <p>
251  * This queries this era using the specified query strategy object.
252  * The {@code TemporalQuery} object defines the logic to be used to
253  * obtain the result. Read the documentation of the query to understand
254  * what the result of this method will be.
255  * <p>
256  * The result of this method is obtained by invoking the
257  * {@link TemporalQuery#queryFrom(TemporalAccessor)} method on the
258  * specified query passing {@code this} as the argument.
259  *
260  * @param <R> the type of the result
261  * @param query the query to invoke, not null
262  * @return the query result, null may be returned (defined by the query)
263  * @throws DateTimeException if unable to query (defined by the query)
264  * @throws ArithmeticException if numeric overflow occurs (defined by the query)
265  */

```

```

266 @SuppressWarnings("unchecked")
267 @Override
268 default <R> R query(TemporalQuery<R> query) {
269     if (query == TemporalQueries.precision()) {
270         return (R) ERAS;
271     }
272     return TemporalAccessor.super.query(query);
273 }
274
275 /**
276  * Adjusts the specified temporal object to have the same era as this object.
277  * <p>
278  * This returns a temporal object of the same observable type as the input
279  * with the era changed to be the same as this.
280  * <p>
281  * The adjustment is equivalent to using {@link Temporal#with(TemporalField, long)}
282  * passing {@link ChronoField#ERA} as the field.
283  * <p>
284  * In most cases, it is clearer to reverse the calling pattern by using
285  * {@link Temporal#with(TemporalAdjuster)}:
286  * <pre>
287  * // these two lines are equivalent, but the second approach is recommended
288  * temporal = thisEra.adjustInto(temporal);
289  * temporal = temporal.with(thisEra);
290  * </pre>
291  * <p>
292  * This instance is immutable and unaffected by this method call.
293  *
294  * @param temporal the target object to be adjusted, not null
295  * @return the adjusted object, not null
296  * @throws DateTimeException if unable to make the adjustment
297  * @throws ArithmeticException if numeric overflow occurs
298  */
299 @Override
300 default Temporal adjustInto(Temporal temporal) {
301     return temporal.with(ERA, getValue());
302 }
303
304 //-----
305 /**
306  * Gets the textual representation of this era.
307  * <p>
308  * This returns the textual name used to identify the era,
309  * suitable for presentation to the user.
310  * The parameters control the style of the returned text and the locale.
311  * <p>
312  * If no textual mapping is found then the {@link #getValue() numeric value} is returned.
313  * <p>
314  * This default implementation is suitable for all implementations.
315  *
316  * @param style the style of the text required, not null
317  * @param locale the locale to use, not null
318  * @return the text value of the era, not null

```



```

319     */
320     default String getDisplayName(TextStyle style, Locale locale) {
321         return new DateTimeFormatterBuilder().appendText(ERA, style).toFormatter(locale).format(
322             this);
323     }
324     // NOTE: methods to convert year-of-era/proleptic-year cannot be here as they may depend on
325     month/day (Japanese)
326 }

```

5.12 HijrahChronology.java

Secuencia 82: java/time/chrono/HijrahChronology.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
28 *
29 * All rights reserved.
30 *
31 * Redistribution and use in source and binary forms, with or without
32 * modification, are permitted provided that the following conditions are met:
33 *
34 * * Redistributions of source code must retain the above copyright notice,
35 *   this list of conditions and the following disclaimer.
36 *
37 * * Redistributions in binary form must reproduce the above copyright notice,
38 *   this list of conditions and the following disclaimer in the documentation
39 *   and/or other materials provided with the distribution.
40 *

```

```
41  * * Neither the name of JSR-310 nor the names of its contributors
42  *   may be used to endorse or promote products derived from this software
43  *   without specific prior written permission.
44  *
45  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56  */
57
58 package java.time.chrono;
59
60 import static java.time.temporal.ChronoField.EPOCH_DAY;
61
62 import java.io.File;
63 import java.io.FileInputStream;
64 import java.io.IOException;
65 import java.io.InputStream;
66 import java.io.InvalidObjectException;
67 import java.io.ObjectInputStream;
68 import java.io.Serializable;
69 import java.security.AccessController;
70 import java.security.PrivilegedActionException;
71 import java.time.Clock;
72 import java.time.DateTimeException;
73 import java.time.Instant;
74 import java.time.LocalDate;
75 import java.time.ZoneId;
76 import java.time.format.ResolverStyle;
77 import java.time.temporal.ChronoField;
78 import java.time.temporal.TemporalAccessor;
79 import java.time.temporal.TemporalField;
80 import java.time.temporal.ValueRange;
81 import java.util.Arrays;
82 import java.util.HashMap;
83 import java.util.List;
84 import java.util.Map;
85 import java.util.Objects;
86 import java.util.Properties;
87
88 import sun.util.logging.PlatformLogger;
89
90 /**
91  * The Hijrah calendar is a lunar calendar supporting Islamic calendars.
92  * <p>
93  * The HijrahChronology follows the rules of the Hijrah calendar system. The Hijrah
```

```

94  * calendar has several variants based on differences in when the new moon is
95  * determined to have occurred and where the observation is made.
96  * In some variants the length of each month is
97  * computed algorithmically from the astronomical data for the moon and earth and
98  * in others the length of the month is determined by an authorized sighting
99  * of the new moon. For the algorithmically based calendars the calendar
100 * can project into the future.
101 * For sighting based calendars only historical data from past
102 * sightings is available.
103 * <p>
104 * The length of each month is 29 or 30 days.
105 * Ordinary years have 354 days; leap years have 355 days.
106 *
107 * <p>
108 * CLDR and LDML identify variants:
109 * <table cellpadding="2" summary="Variants of Hijrah Calendars">
110 * <thead>
111 * <tr class="tableSubHeadingColor">
112 * <th class="colFirst" align="left" >Chronology ID</th>
113 * <th class="colFirst" align="left" >Calendar Type</th>
114 * <th class="colFirst" align="left" >Locale extension, see {@link java.util.Locale}</th>
115 * <th class="colLast" align="left" >Description</th>
116 * </tr>
117 * </thead>
118 * <tbody>
119 * <tr class="altColor">
120 * <td>Hijrah-umalqura</td>
121 * <td>islamic-umalqura</td>
122 * <td>ca-islamic-umalqura</td>
123 * <td>Islamic - Umm Al-Qura calendar of Saudi Arabia</td>
124 * </tr>
125 * </tbody>
126 * </table>
127 * <p>Additional variants may be available through {@link Chronology#getAvailableChronologies()}.
128 *
129 * <p>Example</p>
130 * <p>
131 * Selecting the chronology from the locale uses {@link Chronology#ofLocale}
132 * to find the Chronology based on Locale supported BCP 47 extension mechanism
133 * to request a specific calendar ("ca"). For example,
134 * </p>
135 * <pre>
136 *     Locale locale = Locale.forLanguageTag("en-US-u-ca-islamic-umalqura");
137 *     Chronology chrono = Chronology.ofLocale(locale);
138 * </pre>
139 *
140 * @implSpec
141 * This class is immutable and thread-safe.
142 *
143 * @implNote
144 * Each Hijrah variant is configured individually. Each variant is defined by a
145 * property resource that defines the {@code ID}, the {@code calendar type},
146 * the start of the calendar, the alignment with the

```

```

147 * ISO calendar, and the length of each month for a range of years.
148 * The variants are identified in the {@code calendars.properties} file.
149 * The new properties are prefixed with {@code "calendars.hijrah."}:
150 * <table cellpadding="2" border="0" summary="Configuration of Hijrah Calendar Variants">
151 * <thead>
152 * <tr class="tableSubHeadingColor">
153 * <th class="colFirst" align="left">Property Name</th>
154 * <th class="colFirst" align="left">Property value</th>
155 * <th class="colLast" align="left">Description </th>
156 * </tr>
157 * </thead>
158 * <tbody>
159 * <tr class="altColor">
160 * <td>calendars.hijrah.{ID}</td>
161 * <td>The property resource defining the {@code {ID}} variant</td>
162 * <td>The property resource is located with the {@code calendars.properties} file</td>
163 * </tr>
164 * <tr class="rowColor">
165 * <td>calendars.hijrah.{ID}.type</td>
166 * <td>The calendar type</td>
167 * <td>LDML defines the calendar type names</td>
168 * </tr>
169 * </tbody>
170 * </table>
171 * <p>
172 * The Hijrah property resource is a set of properties that describe the calendar.
173 * The syntax is defined by {@code java.util.Properties#load(Reader)}.
174 * <table cellpadding="2" summary="Configuration of Hijrah Calendar">
175 * <thead>
176 * <tr class="tableSubHeadingColor">
177 * <th class="colFirst" align="left" > Property Name</th>
178 * <th class="colFirst" align="left" > Property value</th>
179 * <th class="colLast" align="left" > Description </th>
180 * </tr>
181 * </thead>
182 * <tbody>
183 * <tr class="altColor">
184 * <td>id</td>
185 * <td>Chronology Id, for example, "Hijrah-umalqura"</td>
186 * <td>The Id of the calendar in common usage</td>
187 * </tr>
188 * <tr class="rowColor">
189 * <td>type</td>
190 * <td>Calendar type, for example, "islamic-umalqura"</td>
191 * <td>LDML defines the calendar types</td>
192 * </tr>
193 * <tr class="altColor">
194 * <td>version</td>
195 * <td>Version, for example: "1.8.0_1"</td>
196 * <td>The version of the Hijrah variant data</td>
197 * </tr>
198 * <tr class="rowColor">
199 * <td>iso-start</td>

```

```

200 * <td>ISO start date, formatted as {@code yyyy-MM-dd}, for example: "1900-04-30"</td>
201 * <td>The ISO date of the first day of the minimum Hijrah year.</td>
202 * </tr>
203 * <tr class="altColor">
204 * <td>yyyy - a numeric 4 digit year, for example "1434"</td>
205 * <td>The value is a sequence of 12 month lengths,
206 * for example: "29 30 29 30 29 30 30 30 29 30 29 29"</td>
207 * <td>The lengths of the 12 months of the year separated by whitespace.
208 * A numeric year property must be present for every year without any gaps.
209 * The month lengths must be between 29-32 inclusive.
210 * </td>
211 * </tr>
212 * </tbody>
213 * </table>
214 *
215 * @since 1.8
216 */
217 public final class HijrahChronology extends AbstractChronology implements Serializable {
218
219     /**
220      * The Hijrah Calendar id.
221      */
222     private final transient String typeId;
223     /**
224      * The Hijrah calendarType.
225      */
226     private final transient String calendarType;
227     /**
228      * Serialization version.
229      */
230     private static final long serialVersionUID = 3127340209035924785L;
231     /**
232      * Singleton instance of the Islamic Umm Al-Qura calendar of Saudi Arabia.
233      * Other Hijrah chronology variants may be available from
234      * {@link Chronology#getAvailableChronologies}.
235      */
236     public static final HijrahChronology INSTANCE;
237     /**
238      * Flag to indicate the initialization of configuration data is complete.
239      * @see #checkCalendarInit()
240      */
241     private transient volatile boolean initComplete;
242     /**
243      * Array of epoch days indexed by Hijrah Epoch month.
244      * Computed by {@link #loadCalendarData}.
245      */
246     private transient int[] hijrahEpochMonthStartDays;
247     /**
248      * The minimum epoch day of this Hijrah calendar.
249      * Computed by {@link #loadCalendarData}.
250      */
251     private transient int minEpochDay;
252     /**

```

```

253     * The maximum epoch day for which calendar data is available.
254     * Computed by {@link #loadCalendarData}.
255     */
256     private transient int maxEpochDay;
257     /**
258     * The minimum epoch month.
259     * Computed by {@link #loadCalendarData}.
260     */
261     private transient int hijrahStartEpochMonth;
262     /**
263     * The minimum length of a month.
264     * Computed by {@link #createEpochMonths}.
265     */
266     private transient int minMonthLength;
267     /**
268     * The maximum length of a month.
269     * Computed by {@link #createEpochMonths}.
270     */
271     private transient int maxMonthLength;
272     /**
273     * The minimum length of a year in days.
274     * Computed by {@link #createEpochMonths}.
275     */
276     private transient int minYearLength;
277     /**
278     * The maximum length of a year in days.
279     * Computed by {@link #createEpochMonths}.
280     */
281     private transient int maxYearLength;
282     /**
283     * A reference to the properties stored in
284     * ${java.home}/lib/calendars.properties
285     */
286     private final transient static Properties calendarProperties;
287
288     /**
289     * Prefix of property names for Hijrah calendar variants.
290     */
291     private static final String PROP_PREFIX = "calendar.hijrah.";
292     /**
293     * Suffix of property names containing the calendar type of a variant.
294     */
295     private static final String PROP_TYPE_SUFFIX = ".type";
296
297     /**
298     * Static initialization of the predefined calendars found in the
299     * lib/calendars.properties file.
300     */
301     static {
302         try {
303             calendarProperties = sun.util.calendar.BaseCalendar.getCalendarProperties();
304         } catch (IOException ioe) {
305             throw new InternalError("Can't initialize lib/calendars.properties", ioe);

```

```

306     }
307
308     try {
309         INSTANCE = new HijrahChronology("Hijrah-umalqura");
310         // Register it by its aliases
311         AbstractChronology.registerChrono(INSTANCE, "Hijrah");
312         AbstractChronology.registerChrono(INSTANCE, "islamic");
313     } catch (DateTimeException ex) {
314         // Absence of Hijrah calendar is fatal to initializing this class.
315         PlatformLogger logger = PlatformLogger.getLogger("java.time.chrono");
316         logger.severe("Unable to initialize Hijrah calendar: Hijrah-umalqura", ex);
317         throw new RuntimeException("Unable to initialize Hijrah-umalqura calendar", ex.getCause());
318     }
319     registerVariants();
320 }
321
322 /**
323  * For each Hijrah variant listed, create the HijrahChronology and register it.
324  * Exceptions during initialization are logged but otherwise ignored.
325  */
326 private static void registerVariants() {
327     for (String name : calendarProperties.stringPropertyNames()) {
328         if (name.startsWith(PROP_PREFIX)) {
329             String id = name.substring(PROP_PREFIX.length());
330             if (id.indexOf('.') >= 0) {
331                 continue; // no name or not a simple name of a calendar
332             }
333             if (id.equals(INSTANCE.getId())) {
334                 continue; // do not duplicate the default
335             }
336             try {
337                 // Create and register the variant
338                 HijrahChronology chrono = new HijrahChronology(id);
339                 AbstractChronology.registerChrono(chrono);
340             } catch (DateTimeException ex) {
341                 // Log error and continue
342                 PlatformLogger logger = PlatformLogger.getLogger("java.time.chrono");
343                 logger.severe("Unable to initialize Hijrah calendar: " + id, ex);
344             }
345         }
346     }
347 }
348
349 /**
350  * Create a HijrahChronology for the named variant.
351  * The resource and calendar type are retrieved from properties
352  * in the {@code calendars.properties}.
353  * The property names are {@code "calendar.hijrah." + id}
354  * and {@code "calendar.hijrah." + id + ".type"}
355  * @param id the id of the calendar
356  * @throws DateTimeException if the calendar type is missing from the properties file.
357  * @throws IllegalArgumentException if the id is empty

```

```

358     */
359     private HijrahChronology(String id) throws DateTimeException {
360         if (id.isEmpty()) {
361             throw new IllegalArgumentException("calendar id is empty");
362         }
363         String propName = PROP_PREFIX + id + PROP_TYPE_SUFFIX;
364         String calType = calendarProperties.getProperty(propName);
365         if (calType == null || calType.isEmpty()) {
366             throw new DateTimeException("calendarType is missing or empty for: " + propName);
367         }
368         this.typeId = id;
369         this.calendarType = calType;
370     }
371
372     /**
373      * Check and ensure that the calendar data has been initialized.
374      * The initialization check is performed at the boundary between
375      * public and package methods. If a public calls another public method
376      * a check is not necessary in the caller.
377      * The constructors of HijrahDate call {@link #getEpochDay} or
378      * {@link #getHijrahDateInfo} so every call from HijrahDate to a
379      * HijrahChronology via package private methods has been checked.
380      *
381      * @throws DateTimeException if the calendar data configuration is
382      *     malformed or IOExceptions occur loading the data
383      */
384     private void checkCalendarInit() {
385         // Keep this short so it can be inlined for performance
386         if (initComplete == false) {
387             loadCalendarData();
388             initComplete = true;
389         }
390     }
391
392     //-----
393     /**
394      * Gets the ID of the chronology.
395      * <p>
396      * The ID uniquely identifies the {@code Chronology}. It can be used to
397      * lookup the {@code Chronology} using {@link Chronology#of(String)}.
398      *
399      * @return the chronology ID, non-null
400      * @see #getCalendarType()
401      */
402     @Override
403     public String getId() {
404         return typeId;
405     }
406
407     /**
408      * Gets the calendar type of the Islamic calendar.
409      * <p>
410      * The calendar type is an identifier defined by the

```



```

411 * <em>Unicode Locale Data Markup Language (LDML)</em> specification.
412 * It can be used to lookup the {@code Chronology} using {@link Chronology#of(String)}.
413 *
414 * @return the calendar system type; non-null if the calendar has
415 *     a standard type, otherwise null
416 * @see #getId()
417 */
418 @Override
419 public String getCalendarType() {
420     return calendarType;
421 }
422
423 //-----
424 /**
425  * Obtains a local date in Hijrah calendar system from the
426  * era, year-of-era, month-of-year and day-of-month fields.
427  *
428  * @param era    the Hijrah era, not null
429  * @param yearOfEra  the year-of-era
430  * @param month   the month-of-year
431  * @param dayOfMonth  the day-of-month
432  * @return the Hijrah local date, not null
433  * @throws DateTimeException if unable to create the date
434  * @throws ClassCastException if the {@code era} is not a {@code HijrahEra}
435  */
436 @Override
437 public HijrahDate date(Era era, int yearOfEra, int month, int dayOfMonth) {
438     return date(prolepticYear(era, yearOfEra), month, dayOfMonth);
439 }
440
441 /**
442  * Obtains a local date in Hijrah calendar system from the
443  * proleptic-year, month-of-year and day-of-month fields.
444  *
445  * @param prolepticYear  the proleptic-year
446  * @param month   the month-of-year
447  * @param dayOfMonth  the day-of-month
448  * @return the Hijrah local date, not null
449  * @throws DateTimeException if unable to create the date
450  */
451 @Override
452 public HijrahDate date(int prolepticYear, int month, int dayOfMonth) {
453     return HijrahDate.of(this, prolepticYear, month, dayOfMonth);
454 }
455
456 /**
457  * Obtains a local date in Hijrah calendar system from the
458  * era, year-of-era and day-of-year fields.
459  *
460  * @param era    the Hijrah era, not null
461  * @param yearOfEra  the year-of-era
462  * @param dayOfYear  the day-of-year
463  * @return the Hijrah local date, not null

```

```

464     * @throws DateTimeException if unable to create the date
465     * @throws ClassCastException if the {@code era} is not a {@code HijrahEra}
466     */
467     @Override
468     public HijrahDate dateYearDay(Era era, int yearOfEra, int dayOfYear) {
469         return dateYearDay(prolepticYear(era, yearOfEra), dayOfYear);
470     }
471
472     /**
473     * Obtains a local date in Hijrah calendar system from the
474     * proleptic-year and day-of-year fields.
475     *
476     * @param prolepticYear the proleptic-year
477     * @param dayOfYear the day-of-year
478     * @return the Hijrah local date, not null
479     * @throws DateTimeException if the value of the year is out of range,
480     * or if the day-of-year is invalid for the year
481     */
482     @Override
483     public HijrahDate dateYearDay(int prolepticYear, int dayOfYear) {
484         HijrahDate date = HijrahDate.of(this, prolepticYear, 1, 1);
485         if (dayOfYear > date.lengthOfYear()) {
486             throw new DateTimeException("Invalid dayOfYear: " + dayOfYear);
487         }
488         return date.plusDays(dayOfYear - 1);
489     }
490
491     /**
492     * Obtains a local date in the Hijrah calendar system from the epoch-day.
493     *
494     * @param epochDay the epoch day
495     * @return the Hijrah local date, not null
496     * @throws DateTimeException if unable to create the date
497     */
498     @Override // override with covariant return type
499     public HijrahDate dateEpochDay(long epochDay) {
500         return HijrahDate.ofEpochDay(this, epochDay);
501     }
502
503     @Override
504     public HijrahDate dateNow() {
505         return dateNow(Clock.systemDefaultZone());
506     }
507
508     @Override
509     public HijrahDate dateNow(ZoneId zone) {
510         return dateNow(Clock.system(zone));
511     }
512
513     @Override
514     public HijrahDate dateNow(Clock clock) {
515         return date(LocalDate.now(clock));
516     }

```

```

517
518     @Override
519     public HijrahDate date(TemporalAccessor temporal) {
520         if (temporal instanceof HijrahDate) {
521             return (HijrahDate) temporal;
522         }
523         return HijrahDate.ofEpochDay(this, temporal.getLong(EPOCH_DAY));
524     }
525
526     @Override
527     @SuppressWarnings("unchecked")
528     public ChronoLocalDateTime<HijrahDate> localDateTime(TemporalAccessor temporal) {
529         return (ChronoLocalDateTime<HijrahDate>) super.localDateTime(temporal);
530     }
531
532     @Override
533     @SuppressWarnings("unchecked")
534     public ChronoZonedDateTime<HijrahDate> zonedDateTime(TemporalAccessor temporal) {
535         return (ChronoZonedDateTime<HijrahDate>) super.zonedDateTime(temporal);
536     }
537
538     @Override
539     @SuppressWarnings("unchecked")
540     public ChronoZonedDateTime<HijrahDate> zonedDateTime(Instant instant, ZoneId zone) {
541         return (ChronoZonedDateTime<HijrahDate>) super.zonedDateTime(instant, zone);
542     }
543
544     //-----
545     @Override
546     public boolean isLeapYear(long prolepticYear) {
547         checkCalendarInit();
548         if (prolepticYear < getMinimumYear() || prolepticYear > getMaximumYear()) {
549             return false;
550         }
551         int len = getYearLength((int) prolepticYear);
552         return (len > 354);
553     }
554
555     @Override
556     public int prolepticYear(Era era, int yearOfEra) {
557         if (era instanceof HijrahEra == false) {
558             throw new ClassCastException("Era must be HijrahEra");
559         }
560         return yearOfEra;
561     }
562
563     @Override
564     public HijrahEra eraOf(int eraValue) {
565         switch (eraValue) {
566             case 1:
567                 return HijrahEra.AH;
568             default:
569                 throw new DateTimeException("invalid Hijrah era");

```

```

570     }
571 }
572
573 @Override
574 public List<Era> eras() {
575     return Arrays.<Era>asList(HijrahEra.values());
576 }
577
578 //-----
579 @Override
580 public ValueRange range(ChronoField field) {
581     checkCalendarInit();
582     if (field instanceof ChronoField) {
583         ChronoField f = field;
584         switch (f) {
585             case DAY_OF_MONTH:
586                 return ValueRange.of(1, 1, getMinimumMonthLength(), getMaximumMonthLength());
587             case DAY_OF_YEAR:
588                 return ValueRange.of(1, getMaximumDayOfYear());
589             case ALIGNED_WEEK_OF_MONTH:
590                 return ValueRange.of(1, 5);
591             case YEAR:
592             case YEAR_OF_ERA:
593                 return ValueRange.of(getMinimumYear(), getMaximumYear());
594             case ERA:
595                 return ValueRange.of(1, 1);
596             default:
597                 return field.range();
598         }
599     }
600     return field.range();
601 }
602
603 //-----
604 @Override // override for return type
605 public HijrahDate resolveDate(Map<TemporalField, Long> fieldValues, ResolverStyle resolverStyle) {
606     return (HijrahDate) super.resolveDate(fieldValues, resolverStyle);
607 }
608
609 //-----
610 /**
611  * Check the validity of a year.
612  *
613  * @param prolepticYear the year to check
614  */
615 int checkValidYear(long prolepticYear) {
616     if (prolepticYear < getMinimumYear() || prolepticYear > getMaximumYear()) {
617         throw new DateTimeException("Invalid Hijrah year: " + prolepticYear);
618     }
619     return (int) prolepticYear;
620 }
621

```

```

622 void checkValidDayOfYear(int dayOfYear) {
623     if (dayOfYear < 1 || dayOfYear > getMaximumDayOfYear()) {
624         throw new DateTimeException("Invalid Hijrah day of year: " + dayOfYear);
625     }
626 }
627
628 void checkValidMonth(int month) {
629     if (month < 1 || month > 12) {
630         throw new DateTimeException("Invalid Hijrah month: " + month);
631     }
632 }
633
634 //-----
635 /**
636  * Returns an array containing the Hijrah year, month and day
637  * computed from the epoch day.
638  *
639  * @param epochDay the EpochDay
640  * @return int[0] = YEAR, int[1] = MONTH, int[2] = DATE
641  */
642 int[] getHijrahDateInfo(int epochDay) {
643     checkCalendarInit(); // ensure that the chronology is initialized
644     if (epochDay < minEpochDay || epochDay >= maxEpochDay) {
645         throw new DateTimeException("Hijrah date out of range");
646     }
647
648     int epochMonth = epochDayToEpochMonth(epochDay);
649     int year = epochMonthToYear(epochMonth);
650     int month = epochMonthToMonth(epochMonth);
651     int day1 = epochMonthToEpochDay(epochMonth);
652     int date = epochDay - day1; // epochDay - dayOfEpoch(year, month);
653
654     int dateInfo[] = new int[3];
655     dateInfo[0] = year;
656     dateInfo[1] = month + 1; // change to 1-based.
657     dateInfo[2] = date + 1; // change to 1-based.
658     return dateInfo;
659 }
660
661 /**
662  * Return the epoch day computed from Hijrah year, month, and day.
663  *
664  * @param prolepticYear the year to represent, 0-origin
665  * @param monthOfYear the month-of-year to represent, 1-origin
666  * @param dayOfMonth the day-of-month to represent, 1-origin
667  * @return the epoch day
668  */
669 long getEpochDay(int prolepticYear, int monthOfYear, int dayOfMonth) {
670     checkCalendarInit(); // ensure that the chronology is initialized
671     checkValidMonth(monthOfYear);
672     int epochMonth = yearToEpochMonth(prolepticYear) + (monthOfYear - 1);
673     if (epochMonth < 0 || epochMonth >= hijrahEpochMonthStartDays.length) {
674         throw new DateTimeException("Invalid Hijrah date, year: " +

```

```

675         prolepticYear + ", month: " + monthOfYear);
676     }
677     if (dayOfMonth < 1 || dayOfMonth > getMonthLength(prolepticYear, monthOfYear)) {
678         throw new DateTimeException("Invalid Hijrah day of month: " + dayOfMonth);
679     }
680     return epochMonthToEpochDay(epochMonth) + (dayOfMonth - 1);
681 }
682
683 /**
684  * Returns day of year for the year and month.
685  *
686  * @param prolepticYear a proleptic year
687  * @param month a month, 1-origin
688  * @return the day of year, 1-origin
689  */
690 int getDayOfYear(int prolepticYear, int month) {
691     return yearMonthToDayOfYear(prolepticYear, (month - 1));
692 }
693
694 /**
695  * Returns month length for the year and month.
696  *
697  * @param prolepticYear a proleptic year
698  * @param monthOfYear a month, 1-origin.
699  * @return the length of the month
700  */
701 int getMonthLength(int prolepticYear, int monthOfYear) {
702     int epochMonth = yearToEpochMonth(prolepticYear) + (monthOfYear - 1);
703     if (epochMonth < 0 || epochMonth >= hijrahEpochMonthStartDays.length) {
704         throw new DateTimeException("Invalid Hijrah date, year: " +
705             prolepticYear + ", month: " + monthOfYear);
706     }
707     return epochMonthLength(epochMonth);
708 }
709
710 /**
711  * Returns year length.
712  * Note: The 12th month must exist in the data.
713  *
714  * @param prolepticYear a proleptic year
715  * @return year length in days
716  */
717 int getYearLength(int prolepticYear) {
718     return yearMonthToDayOfYear(prolepticYear, 12);
719 }
720
721 /**
722  * Return the minimum supported Hijrah year.
723  *
724  * @return the minimum
725  */
726 int getMinimumYear() {
727     return epochMonthToYear(0);

```

```
728     }
729
730     /**
731      * Return the maximum supported Hijrah ear.
732      *
733      * @return the minimum
734      */
735     int getMaximumYear() {
736         return epochMonthToYear(hijrahEpochMonthStartDays.length - 1) - 1;
737     }
738
739     /**
740      * Returns maximum day-of-month.
741      *
742      * @return maximum day-of-month
743      */
744     int getMaximumMonthLength() {
745         return maxMonthLength;
746     }
747
748     /**
749      * Returns smallest maximum day-of-month.
750      *
751      * @return smallest maximum day-of-month
752      */
753     int getMinimumMonthLength() {
754         return minMonthLength;
755     }
756
757     /**
758      * Returns maximum day-of-year.
759      *
760      * @return maximum day-of-year
761      */
762     int getMaximumDayOfYear() {
763         return maxYearLength;
764     }
765
766     /**
767      * Returns smallest maximum day-of-year.
768      *
769      * @return smallest maximum day-of-year
770      */
771     int getSmallestMaximumDayOfYear() {
772         return minYearLength;
773     }
774
775     /**
776      * Returns the epochMonth found by locating the epochDay in the table. The
777      * epochMonth is the index in the table
778      *
779      * @param epochDay
780      * @return The index of the element of the start of the month containing the
```

```
781     * epochDay.
782     */
783     private int epochDayToEpochMonth(int epochDay) {
784         // binary search
785         int ndx = Arrays.binarySearch(hijrahEpochMonthStartDays, epochDay);
786         if (ndx < 0) {
787             ndx = -ndx - 2;
788         }
789         return ndx;
790     }
791
792     /**
793     * Returns the year computed from the epochMonth
794     *
795     * @param epochMonth the epochMonth
796     * @return the Hijrah Year
797     */
798     private int epochMonthToYear(int epochMonth) {
799         return (epochMonth + hijrahStartEpochMonth) / 12;
800     }
801
802     /**
803     * Returns the epochMonth for the Hijrah Year.
804     *
805     * @param year the HijrahYear
806     * @return the epochMonth for the beginning of the year.
807     */
808     private int yearToEpochMonth(int year) {
809         return (year * 12) - hijrahStartEpochMonth;
810     }
811
812     /**
813     * Returns the Hijrah month from the epochMonth.
814     *
815     * @param epochMonth the epochMonth
816     * @return the month of the Hijrah Year
817     */
818     private int epochMonthToMonth(int epochMonth) {
819         return (epochMonth + hijrahStartEpochMonth) % 12;
820     }
821
822     /**
823     * Returns the epochDay for the start of the epochMonth.
824     *
825     * @param epochMonth the epochMonth
826     * @return the epochDay for the start of the epochMonth.
827     */
828     private int epochMonthToEpochDay(int epochMonth) {
829         return hijrahEpochMonthStartDays[epochMonth];
830     }
831 }
832
833 /**
```



```

834     * Returns the day of year for the requested HijrahYear and month.
835     *
836     * @param prolepticYear the Hijrah year
837     * @param month the Hijrah month
838     * @return the day of year for the start of the month of the year
839     */
840     private int yearMonthToDayOfYear(int prolepticYear, int month) {
841         int epochMonthFirst = yearToEpochMonth(prolepticYear);
842         return epochMonthToEpochDay(epochMonthFirst + month)
843             - epochMonthToEpochDay(epochMonthFirst);
844     }
845
846     /**
847     * Returns the length of the epochMonth. It is computed from the start of
848     * the following month minus the start of the requested month.
849     *
850     * @param epochMonth the epochMonth; assumed to be within range
851     * @return the length in days of the epochMonth
852     */
853     private int epochMonthLength(int epochMonth) {
854         // The very last entry in the epochMonth table is not the start of a month
855         return hijrahEpochMonthStartDays[epochMonth + 1]
856             - hijrahEpochMonthStartDays[epochMonth];
857     }
858
859     //-----
860     private static final String KEY_ID = "id";
861     private static final String KEY_TYPE = "type";
862     private static final String KEY_VERSION = "version";
863     private static final String KEY_ISO_START = "iso-start";
864
865     /**
866     * Return the configuration properties from the resource.
867     * <p>
868     * The default location of the variant configuration resource is:
869     * <pre>
870     * "$java.home/lib/" + resource-name
871     * </pre>
872     *
873     * @param resource the name of the calendar property resource
874     * @return a Properties containing the properties read from the resource.
875     * @throws Exception if access to the property resource fails
876     */
877     private static Properties readConfigProperties(final String resource) throws Exception {
878         try {
879             return AccessController
880                 .doPrivileged((java.security.PrivilegedExceptionAction<Properties>)
881                     () -> {
882                         String libDir = System.getProperty("java.home") + File.separator + "lib";
883                         File file = new File(libDir, resource);
884                         Properties props = new Properties();
885                         try (InputStream is = new FileInputStream(file)) {
886                             props.load(is);

```

```

887         }
888         return props;
889     });
890     } catch (PrivilegedActionException pax) {
891         throw pax.getException();
892     }
893 }
894
895 /**
896  * Loads and processes the Hijrah calendar properties file for this calendarType.
897  * The starting Hijrah date and the corresponding ISO date are
898  * extracted and used to calculate the epochDate offset.
899  * The version number is identified and ignored.
900  * Everything else is the data for a year with containing the length of each
901  * of 12 months.
902  *
903  * @throws DateTimeException if initialization of the calendar data from the
904  * resource fails
905  */
906 private void loadCalendarData() {
907     try {
908         String resourceName = calendarProperties.getProperty(PROP_PREFIX + typeId);
909         Objects.requireNonNull(resourceName, "Resource missing for calendar: " + PROP_PREFIX +
910             typeId);
911         Properties props = readConfigProperties(resourceName);
912
913         Map<Integer, int[]> years = new HashMap<>();
914         int minYear = Integer.MAX_VALUE;
915         int maxYear = Integer.MIN_VALUE;
916         String id = null;
917         String type = null;
918         String version = null;
919         int isoStart = 0;
920         for (Map.Entry<Object, Object> entry : props.entrySet()) {
921             String key = (String) entry.getKey();
922             switch (key) {
923                 case KEY_ID:
924                     id = (String) entry.getValue();
925                     break;
926                 case KEY_TYPE:
927                     type = (String) entry.getValue();
928                     break;
929                 case KEY_VERSION:
930                     version = (String) entry.getValue();
931                     break;
932                 case KEY_ISO_START: {
933                     int[] ymd = parseYMD((String) entry.getValue());
934                     isoStart = (int) LocalDate.of(ymd[0], ymd[1], ymd[2]).toEpochDay();
935                     break;
936                 }
937                 default:
938                     try {
939                         // Everything else is either a year or invalid

```

```

939         int year = Integer.valueOf(key);
940         int[] months = parseMonths((String) entry.getValue());
941         years.put(year, months);
942         maxYear = Math.max(maxYear, year);
943         minYear = Math.min(minYear, year);
944     } catch (NumberFormatException nfe) {
945         throw new IllegalArgumentException("bad key: " + key);
946     }
947 }
948 }
949
950 if (!getId().equals(id)) {
951     throw new IllegalArgumentException("Configuration is for a different calendar: " +
952         id);
953 }
954 if (!getCalendarType().equals(type)) {
955     throw new IllegalArgumentException("Configuration is for a different calendar type:
956         " + type);
957 }
958 if (version == null || version.isEmpty()) {
959     throw new IllegalArgumentException("Configuration does not contain a version");
960 }
961 if (isoStart == 0) {
962     throw new IllegalArgumentException("Configuration does not contain a ISO start date
963         ");
964 }
965
966 // Now create and validate the array of epochDays indexed by epochMonth
967 hijrahStartEpochMonth = minYear * 12;
968 minEpochDay = isoStart;
969 hijrahEpochMonthStartDays = createEpochMonths(minEpochDay, minYear, maxYear, years);
970 maxEpochDay = hijrahEpochMonthStartDays[hijrahEpochMonthStartDays.length - 1];
971
972 // Compute the min and max year length in days.
973 for (int year = minYear; year < maxYear; year++) {
974     int length = getYearLength(year);
975     minYearLength = Math.min(minYearLength, length);
976     maxYearLength = Math.max(maxYearLength, length);
977 }
978 } catch (Exception ex) {
979     // Log error and throw a DateTimeException
980     PlatformLogger logger = PlatformLogger.getLogger("java.time.chrono");
981     logger.severe("Unable to initialize Hijrah calendar proxy: " + typeId, ex);
982     throw new DateTimeException("Unable to initialize HijrahCalendar: " + typeId, ex);
983 }
984 }
985
986 /**
987  * Converts the map of year to month lengths ranging from minYear to maxYear
988  * into a linear contiguous array of epochDays. The index is the hijrahMonth
989  * computed from year and month and offset by minYear. The value of each
990  * entry is the epochDay corresponding to the first day of the month.
991  */

```

```

989     * @param minYear The minimum year for which data is provided
990     * @param maxYear The maximum year for which data is provided
991     * @param years a Map of year to the array of 12 month lengths
992     * @return array of epochDays for each month from min to max
993     */
994     private int[] createEpochMonths(int epochDay, int minYear, int maxYear, Map<Integer, int[]>
        years) {
995         // Compute the size for the array of dates
996         int numMonths = (maxYear - minYear + 1) * 12 + 1;
997
998         // Initialize the running epochDay as the corresponding ISO Epoch day
999         int epochMonth = 0; // index into array of epochMonths
1000        int[] epochMonths = new int[numMonths];
1001        minMonthLength = Integer.MAX_VALUE;
1002        maxMonthLength = Integer.MIN_VALUE;
1003
1004        // Only whole years are valid, any zero's in the array are illegal
1005        for (int year = minYear; year <= maxYear; year++) {
1006            int[] months = years.get(year); // must not be gaps
1007            for (int month = 0; month < 12; month++) {
1008                int length = months[month];
1009                epochMonths[epochMonth++] = epochDay;
1010
1011                if (length < 29 || length > 32) {
1012                    throw new IllegalArgumentException("Invalid month length in year: " + minYear);
1013                }
1014                epochDay += length;
1015                minMonthLength = Math.min(minMonthLength, length);
1016                maxMonthLength = Math.max(maxMonthLength, length);
1017            }
1018        }
1019
1020        // Insert the final epochDay
1021        epochMonths[epochMonth++] = epochDay;
1022
1023        if (epochMonth != epochMonths.length) {
1024            throw new IllegalStateException("Did not fill epochMonths exactly: ndx = " + epochMonth
1025                + " should be " + epochMonths.length);
1026        }
1027
1028        return epochMonths;
1029    }
1030
1031    /**
1032     * Parses the 12 months lengths from a property value for a specific year.
1033     *
1034     * @param line the value of a year property
1035     * @return an array of int[12] containing the 12 month lengths
1036     * @throws IllegalArgumentException if the number of months is not 12
1037     * @throws NumberFormatException if the 12 tokens are not numbers
1038     */
1039    private int[] parseMonths(String line) {
1040        int[] months = new int[12];

```

```

1041     String[] numbers = line.split("\\s");
1042     if (numbers.length != 12) {
1043         throw new IllegalArgumentException("wrong number of months on line: " + Arrays.toString
            (numbers) + "; count: " + numbers.length);
1044     }
1045     for (int i = 0; i < 12; i++) {
1046         try {
1047             months[i] = Integer.valueOf(numbers[i]);
1048         } catch (NumberFormatException nfe) {
1049             throw new IllegalArgumentException("bad key: " + numbers[i]);
1050         }
1051     }
1052     return months;
1053 }
1054
1055 /**
1056  * Parse yyyy-MM-dd into a 3 element array [yyyy, mm, dd].
1057  *
1058  * @param string the input string
1059  * @return the 3 element array with year, month, day
1060  */
1061 private int[] parseYMD(String string) {
1062     // yyyy-MM-dd
1063     string = string.trim();
1064     try {
1065         if (string.charAt(4) != '-' || string.charAt(7) != '-') {
1066             throw new IllegalArgumentException("date must be yyyy-MM-dd");
1067         }
1068         int[] ymd = new int[3];
1069         ymd[0] = Integer.valueOf(string.substring(0, 4));
1070         ymd[1] = Integer.valueOf(string.substring(5, 7));
1071         ymd[2] = Integer.valueOf(string.substring(8, 10));
1072         return ymd;
1073     } catch (NumberFormatException ex) {
1074         throw new IllegalArgumentException("date must be yyyy-MM-dd", ex);
1075     }
1076 }
1077
1078 //-----
1079 /**
1080  * Writes the Chronology using a
1081  * <a href="../../serialized-form.html#java.time.chrono.Ser">dedicated serialized form</a>.
1082  * @serialData
1083  * <pre>
1084  *   out.writeByte(1);    // identifies a Chronology
1085  *   out.writeUTF(getId());
1086  * </pre>
1087  *
1088  * @return the instance of {@code Ser}, not null
1089  */
1090 @Override
1091 Object writeReplace() {
1092     return super.writeReplace();

```

```
1093     }
1094
1095     /**
1096      * Defend against malicious streams.
1097      *
1098      * @param s the stream to read
1099      * @throws InvalidObjectException always
1100      */
1101     private void readObject(ObjectInputStream s) throws InvalidObjectException {
1102         throw new InvalidObjectException("Deserialization via serialization delegate");
1103     }
1104 }
```

5.13 HijrahDate.java

Secuencia 83: java/time/chrono/HijrahDate.java

```
1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
28 *
29 * All rights reserved.
30 *
31 * Redistribution and use in source and binary forms, with or without
32 * modification, are permitted provided that the following conditions are met:
33 *
34 * * Redistributions of source code must retain the above copyright notice,
35 *   this list of conditions and the following disclaimer.
36 *
37 * * Redistributions in binary form must reproduce the above copyright notice,
```

```
38 *   this list of conditions and the following disclaimer in the documentation
39 *   and/or other materials provided with the distribution.
40 *
41 * * Neither the name of JSR-310 nor the names of its contributors
42 *   may be used to endorse or promote products derived from this software
43 *   without specific prior written permission.
44 *
45 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56 */
57 package java.time.chrono;
58
59 import static java.time.temporal.ChronoField.ALIGNED_DAY_OF_WEEK_IN_MONTH;
60 import static java.time.temporal.ChronoField.ALIGNED_DAY_OF_WEEK_IN_YEAR;
61 import static java.time.temporal.ChronoField.ALIGNED_WEEK_OF_MONTH;
62 import static java.time.temporal.ChronoField.ALIGNED_WEEK_OF_YEAR;
63 import static java.time.temporal.ChronoField.DAY_OF_MONTH;
64 import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
65 import static java.time.temporal.ChronoField.YEAR;
66
67 import java.io.IOException;
68 import java.io.InvalidObjectException;
69 import java.io.ObjectInput;
70 import java.io.ObjectInputStream;
71 import java.io.ObjectOutput;
72 import java.io.Serializable;
73 import java.time.Clock;
74 import java.time.DateTimeException;
75 import java.time.LocalDate;
76 import java.time.LocalDateTime;
77 import java.time.ZoneId;
78 import java.time.temporal.ChronoField;
79 import java.time.temporal.TemporalAccessor;
80 import java.time.temporal.TemporalAdjuster;
81 import java.time.temporal.TemporalAmount;
82 import java.time.temporal.TemporalField;
83 import java.time.temporal.TemporalQuery;
84 import java.time.temporal.TemporalUnit;
85 import java.time.temporal.UnsupportedTemporalTypeException;
86 import java.time.temporal.ValueRange;
87
88 /**
89 * A date in the Hijrah calendar system.
90 * <p>
```

```

91  * This date operates using one of several variants of the
92  * {@linkplain HijrahChronology Hijrah calendar}.
93  * <p>
94  * The Hijrah calendar has a different total of days in a year than
95  * Gregorian calendar, and the length of each month is based on the period
96  * of a complete revolution of the moon around the earth
97  * (as between successive new moons).
98  * Refer to the {@link HijrahChronology} for details of supported variants.
99  * <p>
100 * Each HijrahDate is created bound to a particular HijrahChronology,
101 * The same chronology is propagated to each HijrahDate computed from the date.
102 * To use a different Hijrah variant, its HijrahChronology can be used
103 * to create new HijrahDate instances.
104 * Alternatively, the {@link #withVariant} method can be used to convert
105 * to a new HijrahChronology.
106 *
107 * <p>
108 * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
109 * class; use of identity-sensitive operations (including reference equality
110 * ({@code ==}), identity hash code, or synchronization) on instances of
111 * {@code HijrahDate} may have unpredictable results and should be avoided.
112 * The {@code equals} method should be used for comparisons.
113 *
114 * @implSpec
115 * This class is immutable and thread-safe.
116 *
117 * @since 1.8
118 */
119 public final class HijrahDate
120     extends ChronoLocalDateImpl<HijrahDate>
121     implements ChronoLocalDate, Serializable {
122
123     /**
124      * Serialization version.
125      */
126     private static final long serialVersionUID = -5207853542612002020L;
127
128     /**
129      * The Chronology of this HijrahDate.
130      */
131     private final transient HijrahChronology chrono;
132
133     /**
134      * The proleptic year.
135      */
136     private final transient int prolepticYear;
137
138     /**
139      * The month-of-year.
140      */
141     private final transient int monthOfYear;
142
143     /**
144      * The day-of-month.
145      */
146     private final transient int dayOfMonth;

```



```

144 //-----
145 /**
146  * Obtains an instance of {@code HijrahDate} from the Hijrah proleptic year,
147  * month-of-year and day-of-month.
148  *
149  * @param prolepticYear the proleptic year to represent in the Hijrah calendar
150  * @param monthOfYear the month-of-year to represent, from 1 to 12
151  * @param dayOfMonth the day-of-month to represent, from 1 to 30
152  * @return the Hijrah date, never null
153  * @throws DateTimeException if the value of any field is out of range
154  */
155 static HijrahDate of(HijrahChronology chrono, int prolepticYear, int monthOfYear, int
    dayOfMonth) {
156     return new HijrahDate(chrono, prolepticYear, monthOfYear, dayOfMonth);
157 }
158
159 /**
160  * Returns a HijrahDate for the chronology and epochDay.
161  * @param chrono The Hijrah chronology
162  * @param epochDay the epoch day
163  * @return a HijrahDate for the epoch day; non-null
164  */
165 static HijrahDate ofEpochDay(HijrahChronology chrono, long epochDay) {
166     return new HijrahDate(chrono, epochDay);
167 }
168
169 //-----
170 /**
171  * Obtains the current {@code HijrahDate} of the Islamic Umm Al-Qura calendar
172  * in the default time-zone.
173  * <p>
174  * This will query the {@link Clock#systemDefaultZone() system clock} in the default
175  * time-zone to obtain the current date.
176  * <p>
177  * Using this method will prevent the ability to use an alternate clock for testing
178  * because the clock is hard-coded.
179  *
180  * @return the current date using the system clock and default time-zone, not null
181  */
182 public static HijrahDate now() {
183     return now(Clock.systemDefaultZone());
184 }
185
186 /**
187  * Obtains the current {@code HijrahDate} of the Islamic Umm Al-Qura calendar
188  * in the specified time-zone.
189  * <p>
190  * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current date.
191  * Specifying the time-zone avoids dependence on the default time-zone.
192  * <p>
193  * Using this method will prevent the ability to use an alternate clock for testing
194  * because the clock is hard-coded.
195  *

```

```

196     * @param zone the zone ID to use, not null
197     * @return the current date using the system clock, not null
198     */
199     public static HijrahDate now(ZoneId zone) {
200         return now(Clock.system(zone));
201     }
202
203     /**
204     * Obtains the current {@code HijrahDate} of the Islamic Umm Al-Qura calendar
205     * from the specified clock.
206     * <p>
207     * This will query the specified clock to obtain the current date - today.
208     * Using this method allows the use of an alternate clock for testing.
209     * The alternate clock may be introduced using {@linkplain Clock dependency injection}.
210     *
211     * @param clock the clock to use, not null
212     * @return the current date, not null
213     * @throws DateTimeException if the current date cannot be obtained
214     */
215     public static HijrahDate now(Clock clock) {
216         return HijrahDate.ofEpochDay(HijrahChronology.INSTANCE, LocalDate.now(clock).toEpochDay());
217     }
218
219     /**
220     * Obtains a {@code HijrahDate} of the Islamic Umm Al-Qura calendar
221     * from the proleptic-year, month-of-year and day-of-month fields.
222     * <p>
223     * This returns a {@code HijrahDate} with the specified fields.
224     * The day must be valid for the year and month, otherwise an exception will be thrown.
225     *
226     * @param prolepticYear the Hijrah proleptic-year
227     * @param month the Hijrah month-of-year, from 1 to 12
228     * @param dayOfMonth the Hijrah day-of-month, from 1 to 30
229     * @return the date in Hijrah calendar system, not null
230     * @throws DateTimeException if the value of any field is out of range,
231     * or if the day-of-month is invalid for the month-year
232     */
233     public static HijrahDate of(int prolepticYear, int month, int dayOfMonth) {
234         return HijrahChronology.INSTANCE.date(prolepticYear, month, dayOfMonth);
235     }
236
237     /**
238     * Obtains a {@code HijrahDate} of the Islamic Umm Al-Qura calendar from a temporal object.
239     * <p>
240     * This obtains a date in the Hijrah calendar system based on the specified temporal.
241     * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
242     * which this factory converts to an instance of {@code HijrahDate}.
243     * <p>
244     * The conversion typically uses the {@link ChronoField#EPOCH_DAY EPOCH_DAY}
245     * field, which is standardized across calendar systems.
246     * <p>
247     * This method matches the signature of the functional interface {@link TemporalQuery}
248     * allowing it to be used as a query via method reference, {@code HijrahDate::from}.

```

```

249     *
250     * @param temporal the temporal object to convert, not null
251     * @return the date in Hijrah calendar system, not null
252     * @throws DateTimeException if unable to convert to a {@code HijrahDate}
253     */
254     public static HijrahDate from(TemporalAccessor temporal) {
255         return HijrahChronology.INSTANCE.date(temporal);
256     }
257
258     //-----
259     /**
260     * Constructs an {@code HijrahDate} with the proleptic-year, month-of-year and
261     * day-of-month fields.
262     *
263     * @param chrono The chronology to create the date with
264     * @param prolepticYear the proleptic year
265     * @param monthOfYear the month of year
266     * @param dayOfMonth the day of month
267     */
268     private HijrahDate(HijrahChronology chrono, int prolepticYear, int monthOfYear, int dayOfMonth)
269     {
270         // Computing the Gregorian day checks the valid ranges
271         chrono.getEpochDay(prolepticYear, monthOfYear, dayOfMonth);
272
273         this.chrono = chrono;
274         this.prolepticYear = prolepticYear;
275         this.monthOfYear = monthOfYear;
276         this.dayOfMonth = dayOfMonth;
277     }
278
279     /**
280     * Constructs an instance with the Epoch Day.
281     *
282     * @param epochDay the epochDay
283     */
284     private HijrahDate(HijrahChronology chrono, long epochDay) {
285         int[] dateInfo = chrono.getHijrahDateInfo((int)epochDay);
286
287         this.chrono = chrono;
288         this.prolepticYear = dateInfo[0];
289         this.monthOfYear = dateInfo[1];
290         this.dayOfMonth = dateInfo[2];
291     }
292
293     //-----
294     /**
295     * Gets the chronology of this date, which is the Hijrah calendar system.
296     * <p>
297     * The {@code Chronology} represents the calendar system in use.
298     * The era and other fields in {@link ChronoField} are defined by the chronology.
299     *
300     * @return the Hijrah chronology, not null
301     */

```

```

301     @Override
302     public HijrahChronology getChronology() {
303         return chrono;
304     }
305
306     /**
307      * Gets the era applicable at this date.
308      * <p>
309      * The Hijrah calendar system has one era, 'AH',
310      * defined by {@link HijrahEra}.
311      *
312      * @return the era applicable at this date, not null
313      */
314     @Override
315     public HijrahEra getEra() {
316         return HijrahEra.AH;
317     }
318
319     /**
320      * Returns the length of the month represented by this date.
321      * <p>
322      * This returns the length of the month in days.
323      * Month lengths in the Hijrah calendar system vary between 29 and 30 days.
324      *
325      * @return the length of the month in days
326      */
327     @Override
328     public int lengthOfMonth() {
329         return chrono.getMonthLength(prolepticYear, monthOfYear);
330     }
331
332     /**
333      * Returns the length of the year represented by this date.
334      * <p>
335      * This returns the length of the year in days.
336      * A Hijrah calendar system year is typically shorter than
337      * that of the ISO calendar system.
338      *
339      * @return the length of the year in days
340      */
341     @Override
342     public int lengthOfYear() {
343         return chrono.getYearLength(prolepticYear);
344     }
345
346     //-----
347     @Override
348     public ValueRange range(TemporalField field) {
349         if (field instanceof ChronoField) {
350             if (isSupported(field)) {
351                 ChronoField f = (ChronoField) field;
352                 switch (f) {
353                     case DAY_OF_MONTH: return ValueRange.of(1, lengthOfMonth());

```

```

354         case DAY_OF_YEAR: return ValueRange.of(1, lengthOfYear());
355         case ALIGNED_WEEK_OF_MONTH: return ValueRange.of(1, 5); // TODO
356         // TODO does the limited range of valid years cause years to
357         // start/end part way through? that would affect range
358     }
359     return getChronology().range(f);
360 }
361 throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
362 }
363 return field.rangeRefinedBy(this);
364 }
365
366 @Override
367 public long getLong(TemporalField field) {
368     if (field instanceof ChronoField) {
369         switch ((ChronoField) field) {
370             case DAY_OF_WEEK: return getDayOfWeek();
371             case ALIGNED_DAY_OF_WEEK_IN_MONTH: return ((getDayOfWeek() - 1) % 7) + 1;
372             case ALIGNED_DAY_OF_WEEK_IN_YEAR: return ((getDayOfYear() - 1) % 7) + 1;
373             case DAY_OF_MONTH: return this.dayOfMonth();
374             case DAY_OF_YEAR: return this.getDayOfYear();
375             case EPOCH_DAY: return toEpochDay();
376             case ALIGNED_WEEK_OF_MONTH: return ((dayOfMonth() - 1) / 7) + 1;
377             case ALIGNED_WEEK_OF_YEAR: return ((getDayOfYear() - 1) / 7) + 1;
378             case MONTH_OF_YEAR: return monthOfYear();
379             case PROLEPTIC_MONTH: return getProlepticMonth();
380             case YEAR_OF_ERA: return prolepticYear();
381             case YEAR: return prolepticYear();
382             case ERA: return getEraValue();
383         }
384         throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
385     }
386     return field.getFrom(this);
387 }
388
389 private long getProlepticMonth() {
390     return prolepticYear * 12L + monthOfYear - 1;
391 }
392
393 @Override
394 public HijrahDate with(TemporalField field, long newValue) {
395     if (field instanceof ChronoField) {
396         ChronoField f = (ChronoField) field;
397         // not using checkValidIntValue so EPOCH_DAY and PROLEPTIC_MONTH work
398         chrono.range(f).checkValidValue(newValue, f); // TODO: validate value
399         int nvalue = (int) newValue;
400         switch (f) {
401             case DAY_OF_WEEK: return plusDays(newValue - getDayOfWeek());
402             case ALIGNED_DAY_OF_WEEK_IN_MONTH: return plusDays(newValue - getLong(
403                 ALIGNED_DAY_OF_WEEK_IN_MONTH));
404             case ALIGNED_DAY_OF_WEEK_IN_YEAR: return plusDays(newValue - getLong(
405                 ALIGNED_DAY_OF_WEEK_IN_YEAR));
406             case DAY_OF_MONTH: return resolvePreviousValid(prolepticYear, monthOfYear, nvalue);

```

```

405         case DAY_OF_YEAR: return plusDays(Math.min(nvalue, lengthOfYear()) - getDayOfYear()
406         );
407         case EPOCH_DAY: return new HijrahDate(chrono, newValue);
408         case ALIGNED_WEEK_OF_MONTH: return plusDays((newValue - getLong(
409             ALIGNED_WEEK_OF_MONTH)) * 7);
410         case ALIGNED_WEEK_OF_YEAR: return plusDays((newValue - getLong(ALIGNED_WEEK_OF_YEAR
411             )) * 7);
412         case MONTH_OF_YEAR: return resolvePreviousValid(prolepticYear, nvalue, dayOfMonth);
413         case PROLEPTIC_MONTH: return plusMonths(newValue - getProlepticMonth());
414         case YEAR_OF_ERA: return resolvePreviousValid(prolepticYear >= 1 ? nvalue : 1 -
415             nvalue, monthOfYear, dayOfMonth);
416         case YEAR: return resolvePreviousValid(nvalue, monthOfYear, dayOfMonth);
417         case ERA: return resolvePreviousValid(1 - prolepticYear, monthOfYear, dayOfMonth);
418     }
419     throw new UnsupportedOperationException("Unsupported field: " + field);
420 }
421 return super.with(field, newValue);
422 }
423
424 private HijrahDate resolvePreviousValid(int prolepticYear, int month, int day) {
425     int monthDays = chrono.getMonthLength(prolepticYear, month);
426     if (day > monthDays) {
427         day = monthDays;
428     }
429     return HijrahDate.of(chrono, prolepticYear, month, day);
430 }
431
432 /**
433  * {@inheritDoc}
434  * @throws DateTimeException if unable to make the adjustment.
435  *     For example, if the adjuster requires an ISO chronology
436  * @throws ArithmeticException {@inheritDoc}
437  */
438 @Override
439 public HijrahDate with(TemporalAdjuster adjuster) {
440     return super.with(adjuster);
441 }
442
443 /**
444  * Returns a {@code HijrahDate} with the Chronology requested.
445  * <p>
446  * The year, month, and day are checked against the new requested
447  * HijrahChronology. If the chronology has a shorter month length
448  * for the month, the day is reduced to be the last day of the month.
449  *
450  * @param chronology the new HijrahChronology, non-null
451  * @return a HijrahDate with the requested HijrahChronology, non-null
452  */
453 public HijrahDate withVariant(HijrahChronology chronology) {
454     if (chrono == chronology) {
455         return this;
456     }
457     // Like resolvePreviousValid the day is constrained to stay in the same month

```

```
454     int monthDays = chronology.getDayOfYear(prolepticYear, monthOfYear);
455     return HijrahDate.of(chronology, prolepticYear, monthOfYear, (dayOfMonth > monthDays) ?
        monthDays : dayOfMonth );
456 }
457
458 /**
459  * {@inheritDoc}
460  * @throws DateTimeException {@inheritDoc}
461  * @throws ArithmeticException {@inheritDoc}
462  */
463 @Override
464 public HijrahDate plus(TemporalAmount amount) {
465     return super.plus(amount);
466 }
467
468 /**
469  * {@inheritDoc}
470  * @throws DateTimeException {@inheritDoc}
471  * @throws ArithmeticException {@inheritDoc}
472  */
473 @Override
474 public HijrahDate minus(TemporalAmount amount) {
475     return super.minus(amount);
476 }
477
478 @Override
479 public long toEpochDay() {
480     return chrono.getEpochDay(prolepticYear, monthOfYear, dayOfMonth);
481 }
482
483 /**
484  * Gets the day-of-year field.
485  * <p>
486  * This method returns the primitive {@code int} value for the day-of-year.
487  *
488  * @return the day-of-year
489  */
490 private int getDayOfYear() {
491     return chrono.getDayOfYear(prolepticYear, monthOfYear) + dayOfMonth;
492 }
493
494 /**
495  * Gets the day-of-week value.
496  *
497  * @return the day-of-week; computed from the epochday
498  */
499 private int getDayOfWeek() {
500     int dow0 = (int) Math.floorMod(toEpochDay() + 3, 7);
501     return dow0 + 1;
502 }
503
504 /**
505  * Gets the Era of this date.
```

```

506     *
507     * @return the Era of this date; computed from epochDay
508     */
509     private int getEraValue() {
510         return (prolepticYear > 1 ? 1 : 0);
511     }
512
513     //-----
514     /**
515      * Checks if the year is a leap year, according to the Hijrah calendar system rules.
516      *
517      * @return true if this date is in a leap year
518      */
519     @Override
520     public boolean isLeapYear() {
521         return chrono.isLeapYear(prolepticYear);
522     }
523
524     //-----
525     @Override
526     HijrahDate plusYears(long years) {
527         if (years == 0) {
528             return this;
529         }
530         int newYear = Math.addExact(this.prolepticYear, (int)years);
531         return resolvePreviousValid(newYear, monthOfYear, dayOfMonth);
532     }
533
534     @Override
535     HijrahDate plusMonths(long monthsToAdd) {
536         if (monthsToAdd == 0) {
537             return this;
538         }
539         long monthCount = prolepticYear * 12L + (monthOfYear - 1);
540         long calcMonths = monthCount + monthsToAdd; // safe overflow
541         int newYear = chrono.checkValidYear(Math.floorDiv(calcMonths, 12L));
542         int newMonth = (int)Math.floorMod(calcMonths, 12L) + 1;
543         return resolvePreviousValid(newYear, newMonth, dayOfMonth);
544     }
545
546     @Override
547     HijrahDate plusWeeks(long weeksToAdd) {
548         return super.plusWeeks(weeksToAdd);
549     }
550
551     @Override
552     HijrahDate plusDays(long days) {
553         return new HijrahDate(chrono, toEpochDay() + days);
554     }
555
556     @Override
557     public HijrahDate plus(long amountToAdd, TemporalUnit unit) {
558         return super.plus(amountToAdd, unit);

```



```
559     }
560
561     @Override
562     public HijrahDate minus(long amountToSubtract, TemporalUnit unit) {
563         return super.minus(amountToSubtract, unit);
564     }
565
566     @Override
567     HijrahDate minusYears(long yearsToSubtract) {
568         return super.minusYears(yearsToSubtract);
569     }
570
571     @Override
572     HijrahDate minusMonths(long monthsToSubtract) {
573         return super.minusMonths(monthsToSubtract);
574     }
575
576     @Override
577     HijrahDate minusWeeks(long weeksToSubtract) {
578         return super.minusWeeks(weeksToSubtract);
579     }
580
581     @Override
582     HijrahDate minusDays(long daysToSubtract) {
583         return super.minusDays(daysToSubtract);
584     }
585
586     @Override          // for javadoc and covariant return type
587     @SuppressWarnings("unchecked")
588     public final ChronoLocalDateTime<HijrahDate> atTime(LocalTime localTime) {
589         return (ChronoLocalDateTime<HijrahDate>)super.atTime(localTime);
590     }
591
592     @Override
593     public ChronoPeriod until(ChronoLocalDate endDate) {
594         // TODO: untested
595         HijrahDate end = getChronology().date(endDate);
596         long totalMonths = (end.prolepticYear - this.prolepticYear) * 12 + (end.monthOfYear - this.
597             monthOfYear); // safe
598         int days = end.dayOfMonth - this.dayOfMonth;
599         if (totalMonths > 0 && days < 0) {
600             totalMonths--;
601             HijrahDate calcDate = this.plusMonths(totalMonths);
602             days = (int) (end.toEpochDay() - calcDate.toEpochDay()); // safe
603         } else if (totalMonths < 0 && days > 0) {
604             totalMonths++;
605             days -= end.lengthOfMonth();
606         }
607         long years = totalMonths / 12; // safe
608         int months = (int) (totalMonths % 12); // safe
609         return getChronology().period(Math.toIntExact(years), months, days);
610     }
```

```

611 //-----
612 /**
613  * Compares this date to another date, including the chronology.
614  * <p>
615  * Compares this {@code HijrahDate} with another ensuring that the date is the same.
616  * <p>
617  * Only objects of type {@code HijrahDate} are compared, other types return false.
618  * To compare the dates of two {@code TemporalAccessor} instances, including dates
619  * in two different chronologies, use {@link ChronoField#EPOCH_DAY} as a comparator.
620  *
621  * @param obj the object to check, null returns false
622  * @return true if this is equal to the other date and the Chronologies are equal
623  */
624 @Override // override for performance
625 public boolean equals(Object obj) {
626     if (this == obj) {
627         return true;
628     }
629     if (obj instanceof HijrahDate) {
630         HijrahDate otherDate = (HijrahDate) obj;
631         return prolepticYear == otherDate.prolepticYear
632             && this.monthOfYear == otherDate.monthOfYear
633             && this.dayOfMonth == otherDate.dayOfMonth
634             && getChronology().equals(otherDate.getChronology());
635     }
636     return false;
637 }
638
639 /**
640  * A hash code for this date.
641  *
642  * @return a suitable hash code based only on the Chronology and the date
643  */
644 @Override // override for performance
645 public int hashCode() {
646     int yearValue = prolepticYear;
647     int monthValue = monthOfYear;
648     int dayValue = dayOfMonth;
649     return getChronology().getId().hashCode() ^ (yearValue & 0xFFFFF800)
650         ^ ((yearValue << 11) + (monthValue << 6) + (dayValue));
651 }
652
653 //-----
654 /**
655  * Defend against malicious streams.
656  *
657  * @param s the stream to read
658  * @throws InvalidObjectException always
659  */
660 private void readObject(ObjectInputStream s) throws InvalidObjectException {
661     throw new InvalidObjectException("Deserialization via serialization delegate");
662 }
663

```

```

664  /**
665   * Writes the object using a
666   * <a href="../../serialized-form.html#java.time.chrono.Ser">dedicated serialized form</a>.
667   * @serialData
668   * <pre>
669   *   out.writeByte(6);           // identifies a HijrahDate
670   *   out.writeObject(chrono);    // the HijrahChronology variant
671   *   out.writeInt(get(YEAR));
672   *   out.writeByte(get(MONTH_OF_YEAR));
673   *   out.writeByte(get(DAY_OF_MONTH));
674   * </pre>
675   *
676   * @return the instance of {@code Ser}, not null
677   */
678  private Object writeReplace() {
679      return new Ser(Ser.HIJRAH_DATE_TYPE, this);
680  }
681
682  void writeExternal(ObjectOutput out) throws IOException {
683      // HijrahChronology is implicit in the Hijrah_DATE_TYPE
684      out.writeObject(getChronology());
685      out.writeInt(get(YEAR));
686      out.writeByte(get(MONTH_OF_YEAR));
687      out.writeByte(get(DAY_OF_MONTH));
688  }
689
690  static HijrahDate readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
691      HijrahChronology chrono = (HijrahChronology) in.readObject();
692      int year = in.readInt();
693      int month = in.readByte();
694      int dayOfMonth = in.readByte();
695      return chrono.date(year, month, dayOfMonth);
696  }
697
698  }

```

5.14 HijrahEra.java

Secuencia 84: java/time/chrono/HijrahEra.java

```

1  /**
2   * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3   * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4   *
5   *
6   *
7   *
8   *
9   *
10  *
11  *
12  *
13  *
14  *

```

```
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.chrono;
63
64  import static java.time.temporal.ChronoField.ERA;
65
66  import java.time.DateTimeException;
67  import java.time.temporal.ChronoField;
```

```

68 import java.time.temporal.TemporalField;
69 import java.time.temporal.UnsupportedTemporalTypeException;
70 import java.time.temporal.ValueRange;
71
72 /**
73  * An era in the Hijrah calendar system.
74  * <p>
75  * The Hijrah calendar system has only one era covering the
76  * proleptic years greater than zero.
77  * <p>
78  * <b>Do not use {@code ordinal()} to obtain the numeric representation of {@code HijrahEra}.
79  * Use {@code getValue()} instead.</b>
80  *
81  * @implSpec
82  * This is an immutable and thread-safe enum.
83  *
84  * @since 1.8
85  */
86 public enum HijrahEra implements Era {
87
88     /**
89      * The singleton instance for the current era, 'Anno Hegirae',
90      * which has the numeric value 1.
91      */
92     AH;
93
94     //-----
95     /**
96      * Obtains an instance of {@code HijrahEra} from an {@code int} value.
97      * <p>
98      * The current era, which is the only accepted value, has the value 1
99      *
100     * @param hijrahEra the era to represent, only 1 supported
101     * @return the HijrahEra.AH singleton, not null
102     * @throws DateTimeException if the value is invalid
103     */
104     public static HijrahEra of(int hijrahEra) {
105         if (hijrahEra == 1 ) {
106             return AH;
107         } else {
108             throw new DateTimeException("Invalid era: " + hijrahEra);
109         }
110     }
111
112     //-----
113     /**
114      * Gets the numeric era {@code int} value.
115      * <p>
116      * The era AH has the value 1.
117      *
118      * @return the era value, 1 (AH)
119      */
120     @Override

```

```

121     public int getValue() {
122         return 1;
123     }
124
125     //-----
126     /**
127      * Gets the range of valid values for the specified field.
128      * <p>
129      * The range object expresses the minimum and maximum valid values for a field.
130      * This era is used to enhance the accuracy of the returned range.
131      * If it is not possible to return the range, because the field is not supported
132      * or for some other reason, an exception is thrown.
133      * <p>
134      * If the field is a {@link ChronoField} then the query is implemented here.
135      * The {@code ERA} field returns the range.
136      * All other {@code ChronoField} instances will throw an {@code
137          UnsupportedTemporalTypeException}.
138      * <p>
139      * If the field is not a {@code ChronoField}, then the result of this method
140      * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
141      * passing {@code this} as the argument.
142      * Whether the range can be obtained is determined by the field.
143      * <p>
144      * The {@code ERA} field returns a range for the one valid Hijrah era.
145      *
146      * @param field the field to query the range for, not null
147      * @return the range of valid values for the field, not null
148      * @throws DateTimeException if the range for the field cannot be obtained
149      * @throws UnsupportedTemporalTypeException if the unit is not supported
150      */
151     @Override // override as super would return range from 0 to 1
152     public ValueRange range(TemporalField field) {
153         if (field == ERA) {
154             return ValueRange.of(1, 1);
155         }
156         return Era.super.range(field);
157     }
158 }

```

5.15 IsoChronology.java

Secuencia 85: java/time/chrono/IsoChronology.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *

```

```
11  *
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.chrono;
63
```

```
64 import java.io.InvalidObjectException;
65 import static java.time.temporal.ChronoField.DAY_OF_MONTH;
66 import static java.time.temporal.ChronoField.ERA;
67 import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
68 import static java.time.temporal.ChronoField.PROLEPTIC_MONTH;
69 import static java.time.temporal.ChronoField.YEAR;
70 import static java.time.temporal.ChronoField.YEAR_OF_ERA;
71
72 import java.io.ObjectInputStream;
73 import java.io.Serializable;
74 import java.time.Clock;
75 import java.time.DateTimeException;
76 import java.time.Instant;
77 import java.time.LocalDate;
78 import java.time.LocalDateTime;
79 import java.time.Month;
80 import java.time.Period;
81 import java.time.Year;
82 import java.time.ZoneId;
83 import java.time.ZonedDateTime;
84 import java.time.format.ResolverStyle;
85 import java.time.temporal.ChronoField;
86 import java.time.temporal.TemporalAccessor;
87 import java.time.temporal.TemporalField;
88 import java.time.temporal.ValueRange;
89 import java.util.Arrays;
90 import java.util.List;
91 import java.util.Locale;
92 import java.util.Map;
93 import java.util.Objects;
94
95 /**
96  * The ISO calendar system.
97  * <p>
98  * This chronology defines the rules of the ISO calendar system.
99  * This calendar system is based on the ISO-8601 standard, which is the
100  * <i>de facto</i> world calendar.
101  * <p>
102  * The fields are defined as follows:
103  * <ul>
104  * <li>era - There are two eras, 'Current Era' (CE) and 'Before Current Era' (BCE).
105  * <li>year-of-era - The year-of-era is the same as the proleptic-year for the current CE era.
106  * For the BCE era before the ISO epoch the year increases from 1 upwards as time goes backwards.
107  * <li>proleptic-year - The proleptic year is the same as the year-of-era for the
108  * current era. For the previous era, years have zero, then negative values.
109  * <li>month-of-year - There are 12 months in an ISO year, numbered from 1 to 12.
110  * <li>day-of-month - There are between 28 and 31 days in each of the ISO month, numbered from 1 to
111  * 31.
112  * Months 4, 6, 9 and 11 have 30 days, Months 1, 3, 5, 7, 8, 10 and 12 have 31 days.
113  * Month 2 has 28 days, or 29 in a leap year.
114  * <li>day-of-year - There are 365 days in a standard ISO year and 366 in a leap year.
115  * The days are numbered from 1 to 365 or 1 to 366.
116  * <li>leap-year - Leap years occur every 4 years, except where the year is divisible by 100 and not
```



```

        divisble by 400.
116 * </ul>
117 *
118 * @implSpec
119 * This class is immutable and thread-safe.
120 *
121 * @since 1.8
122 */
123 public final class IsoChronology extends AbstractChronology implements Serializable {
124
125     /**
126      * Singleton instance of the ISO chronology.
127      */
128     public static final IsoChronology INSTANCE = new IsoChronology();
129
130     /**
131      * Serialization version.
132      */
133     private static final long serialVersionUID = -1440403870442975015L;
134
135     /**
136      * Restricted constructor.
137      */
138     private IsoChronology() {
139     }
140
141     //-----
142     /**
143      * Gets the ID of the chronology - 'ISO'.
144      * <p>
145      * The ID uniquely identifies the {@code Chronology}.
146      * It can be used to lookup the {@code Chronology} using {@link Chronology#of(String)}.
147      *
148      * @return the chronology ID - 'ISO'
149      * @see #getCalendarType()
150      */
151     @Override
152     public String getId() {
153         return "ISO";
154     }
155
156     /**
157      * Gets the calendar type of the underlying calendar system - 'iso8601'.
158      * <p>
159      * The calendar type is an identifier defined by the
160      * <em>Unicode Locale Data Markup Language (LDML)</em> specification.
161      * It can be used to lookup the {@code Chronology} using {@link Chronology#of(String)}.
162      * It can also be used as part of a locale, accessible via
163      * {@link Locale#getUnicodeLocaleType(String)} with the key 'ca'.
164      *
165      * @return the calendar system type - 'iso8601'
166      * @see #getId()
167      */

```

```

168     @Override
169     public String getCalendarType() {
170         return "iso8601";
171     }
172
173     //-----
174     /**
175      * Obtains an ISO local date from the era, year-of-era, month-of-year
176      * and day-of-month fields.
177      *
178      * @param era    the ISO era, not null
179      * @param yearOfEra  the ISO year-of-era
180      * @param month    the ISO month-of-year
181      * @param dayOfMonth  the ISO day-of-month
182      * @return the ISO local date, not null
183      * @throws DateTimeException if unable to create the date
184      * @throws ClassCastException if the type of {@code era} is not {@code IsoEra}
185      */
186     @Override // override with covariant return type
187     public LocalDate date(Era era, int yearOfEra, int month, int dayOfMonth) {
188         return date(prolepticYear(era, yearOfEra), month, dayOfMonth);
189     }
190
191     /**
192      * Obtains an ISO local date from the proleptic-year, month-of-year
193      * and day-of-month fields.
194      * <p>
195      * This is equivalent to {@link LocalDate#of(int, int, int)}.
196      *
197      * @param prolepticYear  the ISO proleptic-year
198      * @param month    the ISO month-of-year
199      * @param dayOfMonth  the ISO day-of-month
200      * @return the ISO local date, not null
201      * @throws DateTimeException if unable to create the date
202      */
203     @Override // override with covariant return type
204     public LocalDate date(int prolepticYear, int month, int dayOfMonth) {
205         return LocalDate.of(prolepticYear, month, dayOfMonth);
206     }
207
208     /**
209      * Obtains an ISO local date from the era, year-of-era and day-of-year fields.
210      *
211      * @param era    the ISO era, not null
212      * @param yearOfEra  the ISO year-of-era
213      * @param dayOfYear  the ISO day-of-year
214      * @return the ISO local date, not null
215      * @throws DateTimeException if unable to create the date
216      */
217     @Override // override with covariant return type
218     public LocalDate dateYearDay(Era era, int yearOfEra, int dayOfYear) {
219         return dateYearDay(prolepticYear(era, yearOfEra), dayOfYear);
220     }

```

```

221
222 /**
223  * Obtains an ISO local date from the proleptic-year and day-of-year fields.
224  * <p>
225  * This is equivalent to {@link LocalDate#ofYearDay(int, int)}.
226  *
227  * @param prolepticYear the ISO proleptic-year
228  * @param dayOfYear the ISO day-of-year
229  * @return the ISO local date, not null
230  * @throws DateTimeException if unable to create the date
231  */
232 @Override // override with covariant return type
233 public LocalDate dateYearDay(int prolepticYear, int dayOfYear) {
234     return LocalDate.ofYearDay(prolepticYear, dayOfYear);
235 }
236
237 /**
238  * Obtains an ISO local date from the epoch-day.
239  * <p>
240  * This is equivalent to {@link LocalDate#ofEpochDay(long)}.
241  *
242  * @param epochDay the epoch day
243  * @return the ISO local date, not null
244  * @throws DateTimeException if unable to create the date
245  */
246 @Override // override with covariant return type
247 public LocalDate dateEpochDay(long epochDay) {
248     return LocalDate.ofEpochDay(epochDay);
249 }
250
251 //-----
252 /**
253  * Obtains an ISO local date from another date-time object.
254  * <p>
255  * This is equivalent to {@link LocalDate#from(TemporalAccessor)}.
256  *
257  * @param temporal the date-time object to convert, not null
258  * @return the ISO local date, not null
259  * @throws DateTimeException if unable to create the date
260  */
261 @Override // override with covariant return type
262 public LocalDate date(TemporalAccessor temporal) {
263     return LocalDate.from(temporal);
264 }
265
266 /**
267  * Obtains an ISO local date-time from another date-time object.
268  * <p>
269  * This is equivalent to {@link LocalDateTime#from(TemporalAccessor)}.
270  *
271  * @param temporal the date-time object to convert, not null
272  * @return the ISO local date-time, not null
273  * @throws DateTimeException if unable to create the date-time

```

```

274     */
275     @Override // override with covariant return type
276     public LocalDateTime localDateTime(TemporalAccessor temporal) {
277         return LocalDateTime.from(temporal);
278     }
279
280     /**
281     * Obtains an ISO zoned date-time from another date-time object.
282     * <p>
283     * This is equivalent to {@link ZonedDateTime#from(TemporalAccessor)}.
284     *
285     * @param temporal the date-time object to convert, not null
286     * @return the ISO zoned date-time, not null
287     * @throws DateTimeException if unable to create the date-time
288     */
289     @Override // override with covariant return type
290     public ZonedDateTime zonedDateTime(TemporalAccessor temporal) {
291         return ZonedDateTime.from(temporal);
292     }
293
294     /**
295     * Obtains an ISO zoned date-time in this chronology from an {@code Instant}.
296     * <p>
297     * This is equivalent to {@link ZonedDateTime#ofInstant(Instant, ZoneId)}.
298     *
299     * @param instant the instant to create the date-time from, not null
300     * @param zone the time-zone, not null
301     * @return the zoned date-time, not null
302     * @throws DateTimeException if the result exceeds the supported range
303     */
304     @Override
305     public ZonedDateTime zonedDateTime(Instant instant, ZoneId zone) {
306         return ZonedDateTime.ofInstant(instant, zone);
307     }
308
309     //-----
310     /**
311     * Obtains the current ISO local date from the system clock in the default time-zone.
312     * <p>
313     * This will query the {@link Clock#systemDefaultZone() system clock} in the default
314     * time-zone to obtain the current date.
315     * <p>
316     * Using this method will prevent the ability to use an alternate clock for testing
317     * because the clock is hard-coded.
318     *
319     * @return the current ISO local date using the system clock and default time-zone, not null
320     * @throws DateTimeException if unable to create the date
321     */
322     @Override // override with covariant return type
323     public LocalDate dateNow() {
324         return dateNow(Clock.systemDefaultZone());
325     }
326

```

```

327 /**
328  * Obtains the current ISO local date from the system clock in the specified time-zone.
329  * <p>
330  * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current date.
331  * Specifying the time-zone avoids dependence on the default time-zone.
332  * <p>
333  * Using this method will prevent the ability to use an alternate clock for testing
334  * because the clock is hard-coded.
335  *
336  * @return the current ISO local date using the system clock, not null
337  * @throws DateTimeException if unable to create the date
338  */
339 @Override // override with covariant return type
340 public LocalDate dateNow(ZoneId zone) {
341     return dateNow(Clock.system(zone));
342 }
343
344 /**
345  * Obtains the current ISO local date from the specified clock.
346  * <p>
347  * This will query the specified clock to obtain the current date - today.
348  * Using this method allows the use of an alternate clock for testing.
349  * The alternate clock may be introduced using {@link Clock dependency injection}.
350  *
351  * @param clock the clock to use, not null
352  * @return the current ISO local date, not null
353  * @throws DateTimeException if unable to create the date
354  */
355 @Override // override with covariant return type
356 public LocalDate dateNow(Clock clock) {
357     Objects.requireNonNull(clock, "clock");
358     return date(LocalDate.now(clock));
359 }
360
361 //-----
362 /**
363  * Checks if the year is a leap year, according to the ISO proleptic
364  * calendar system rules.
365  * <p>
366  * This method applies the current rules for leap years across the whole time-line.
367  * In general, a year is a leap year if it is divisible by four without
368  * remainder. However, years divisible by 100, are not leap years, with
369  * the exception of years divisible by 400 which are.
370  * <p>
371  * For example, 1904 is a leap year it is divisible by 4.
372  * 1900 was not a leap year as it is divisible by 100, however 2000 was a
373  * leap year as it is divisible by 400.
374  * <p>
375  * The calculation is proleptic - applying the same rules into the far future and far past.
376  * This is historically inaccurate, but is correct for the ISO-8601 standard.
377  *
378  * @param prolepticYear the ISO proleptic year to check
379  * @return true if the year is leap, false otherwise

```

```

380     */
381     @Override
382     public boolean isLeapYear(long prolepticYear) {
383         return ((prolepticYear & 3) == 0) && ((prolepticYear % 100) != 0 || (prolepticYear % 400)
384             == 0);
385     }
386     @Override
387     public int prolepticYear(Era era, int yearOfEra) {
388         if (era instanceof IsoEra == false) {
389             throw new ClassCastException("Era must be IsoEra");
390         }
391         return (era == IsoEra.CE ? yearOfEra : 1 - yearOfEra);
392     }
393     @Override
394     public IsoEra eraOf(int eraValue) {
395         return IsoEra.of(eraValue);
396     }
397     @Override
398     public List<Era> eras() {
399         return Arrays.<Era>asList(IsoEra.values());
400     }
401     //-----
402     /**
403      * Resolves parsed {@code ChronoField} values into a date during parsing.
404      * <p>
405      * Most {@code TemporalField} implementations are resolved using the
406      * resolve method on the field. By contrast, the {@code ChronoField} class
407      * defines fields that only have meaning relative to the chronology.
408      * As such, {@code ChronoField} date fields are resolved here in the
409      * context of a specific chronology.
410      * <p>
411      * {@code ChronoField} instances on the ISO calendar system are resolved
412      * as follows.
413      * <ul>
414      * <li>{@code EPOCH_DAY} - If present, this is converted to a {@code LocalDate}
415      *   and all other date fields are then cross-checked against the date.
416      * <li>{@code PROLEPTIC_MONTH} - If present, then it is split into the
417      *   {@code YEAR} and {@code MONTH_OF_YEAR}. If the mode is strict or smart
418      *   then the field is validated.
419      * <li>{@code YEAR_OF_ERA} and {@code ERA} - If both are present, then they
420      *   are combined to form a {@code YEAR}. In lenient mode, the {@code YEAR_OF_ERA}
421      *   range is not validated, in smart and strict mode it is. The {@code ERA} is
422      *   validated for range in all three modes. If only the {@code YEAR_OF_ERA} is
423      *   present, and the mode is smart or lenient, then the current era (CE/AD)
424      *   is assumed. In strict mode, no era is assumed and the {@code YEAR_OF_ERA} is
425      *   left untouched. If only the {@code ERA} is present, then it is left untouched.
426      * <li>{@code YEAR}, {@code MONTH_OF_YEAR} and {@code DAY_OF_MONTH} -
427      *   If all three are present, then they are combined to form a {@code LocalDate}.
428      *   In all three modes, the {@code YEAR} is validated. If the mode is smart or strict,

```

```

432 * then the month and day are validated, with the day validated from 1 to 31.
433 * If the mode is lenient, then the date is combined in a manner equivalent to
434 * creating a date on the first of January in the requested year, then adding
435 * the difference in months, then the difference in days.
436 * If the mode is smart, and the day-of-month is greater than the maximum for
437 * the year-month, then the day-of-month is adjusted to the last day-of-month.
438 * If the mode is strict, then the three fields must form a valid date.
439 * <li>{@code YEAR} and {@code DAY_OF_YEAR} -
440 * If both are present, then they are combined to form a {@code LocalDate}.
441 * In all three modes, the {@code YEAR} is validated.
442 * If the mode is lenient, then the date is combined in a manner equivalent to
443 * creating a date on the first of January in the requested year, then adding
444 * the difference in days.
445 * If the mode is smart or strict, then the two fields must form a valid date.
446 * <li>{@code YEAR}, {@code MONTH_OF_YEAR}, {@code ALIGNED_WEEK_OF_MONTH} and
447 * {@code ALIGNED_DAY_OF_WEEK_IN_MONTH} -
448 * If all four are present, then they are combined to form a {@code LocalDate}.
449 * In all three modes, the {@code YEAR} is validated.
450 * If the mode is lenient, then the date is combined in a manner equivalent to
451 * creating a date on the first of January in the requested year, then adding
452 * the difference in months, then the difference in weeks, then in days.
453 * If the mode is smart or strict, then the all four fields are validated to
454 * their outer ranges. The date is then combined in a manner equivalent to
455 * creating a date on the first day of the requested year and month, then adding
456 * the amount in weeks and days to reach their values. If the mode is strict,
457 * the date is additionally validated to check that the day and week adjustment
458 * did not change the month.
459 * <li>{@code YEAR}, {@code MONTH_OF_YEAR}, {@code ALIGNED_WEEK_OF_MONTH} and
460 * {@code DAY_OF_WEEK} - If all four are present, then they are combined to
461 * form a {@code LocalDate}. The approach is the same as described above for
462 * years, months and weeks in {@code ALIGNED_DAY_OF_WEEK_IN_MONTH}.
463 * The day-of-week is adjusted as the next or same matching day-of-week once
464 * the years, months and weeks have been handled.
465 * <li>{@code YEAR}, {@code ALIGNED_WEEK_OF_YEAR} and {@code ALIGNED_DAY_OF_WEEK_IN_YEAR} -
466 * If all three are present, then they are combined to form a {@code LocalDate}.
467 * In all three modes, the {@code YEAR} is validated.
468 * If the mode is lenient, then the date is combined in a manner equivalent to
469 * creating a date on the first of January in the requested year, then adding
470 * the difference in weeks, then in days.
471 * If the mode is smart or strict, then the all three fields are validated to
472 * their outer ranges. The date is then combined in a manner equivalent to
473 * creating a date on the first day of the requested year, then adding
474 * the amount in weeks and days to reach their values. If the mode is strict,
475 * the date is additionally validated to check that the day and week adjustment
476 * did not change the year.
477 * <li>{@code YEAR}, {@code ALIGNED_WEEK_OF_YEAR} and {@code DAY_OF_WEEK} -
478 * If all three are present, then they are combined to form a {@code LocalDate}.
479 * The approach is the same as described above for years and weeks in
480 * {@code ALIGNED_DAY_OF_WEEK_IN_YEAR}. The day-of-week is adjusted as the
481 * next or same matching day-of-week once the years and weeks have been handled.
482 * </ul>
483 *
484 * @param fieldValues the map of fields to values, which can be updated, not null

```

```

485     * @param resolverStyle the requested type of resolve, not null
486     * @return the resolved date, null if insufficient information to create a date
487     * @throws DateTimeException if the date cannot be resolved, typically
488     * because of a conflict in the input data
489     */
490     @Override // override for performance
491     public LocalDate resolveDate(Map<TemporalField, Long> fieldValues, ResolverStyle resolverStyle)
492     {
493         return (LocalDate) super.resolveDate(fieldValues, resolverStyle);
494     }
495
496     @Override // override for better proleptic algorithm
497     void resolveProlepticMonth(Map<TemporalField, Long> fieldValues, ResolverStyle resolverStyle) {
498         Long pMonth = fieldValues.remove(PROLEPTIC_MONTH);
499         if (pMonth != null) {
500             if (resolverStyle != ResolverStyle.LENIENT) {
501                 PROLEPTIC_MONTH.checkValidValue(pMonth);
502             }
503             addFieldValue(fieldValues, MONTH_OF_YEAR, Math.floorMod(pMonth, 12) + 1);
504             addFieldValue(fieldValues, YEAR, Math.floorDiv(pMonth, 12));
505         }
506     }
507
508     @Override // override for enhanced behaviour
509     LocalDate resolveYearOfEra(Map<TemporalField, Long> fieldValues, ResolverStyle resolverStyle) {
510         Long yoeLong = fieldValues.remove(YEAR_OF_ERA);
511         if (yoeLong != null) {
512             if (resolverStyle != ResolverStyle.LENIENT) {
513                 YEAR_OF_ERA.checkValidValue(yoeLong);
514             }
515             Long era = fieldValues.remove(ERA);
516             if (era == null) {
517                 Long year = fieldValues.get(YEAR);
518                 if (resolverStyle == ResolverStyle.STRICT) {
519                     // do not invent era if strict, but do cross-check with year
520                     if (year != null) {
521                         addFieldValue(fieldValues, YEAR, (year > 0 ? yoeLong: Math.subtractExact(1,
522                             yoeLong)));
523                     } else {
524                         // reinstate the field removed earlier, no cross-check issues
525                         fieldValues.put(YEAR_OF_ERA, yoeLong);
526                     }
527                 } else {
528                     // invent era
529                     addFieldValue(fieldValues, YEAR, (year == null || year > 0 ? yoeLong: Math.
530                         subtractExact(1, yoeLong)));
531                 }
532             } else if (era.longValue() == 1L) {
533                 addFieldValue(fieldValues, YEAR, yoeLong);
534             } else if (era.longValue() == 0L) {
535                 addFieldValue(fieldValues, YEAR, Math.subtractExact(1, yoeLong));
536             } else {
537                 throw new DateTimeException("Invalid value for era: " + era);
538             }
539         }
540     }

```



```

535     }
536   } else if (fieldValues.containsKey(ERA)) {
537     ERA.checkValidValue(fieldValues.get(ERA)); // always validated
538   }
539   return null;
540 }
541
542 @Override // override for performance
543 LocalDate resolveYMD(Map <TemporalField, Long> fieldValues, ResolverStyle resolverStyle) {
544   int y = YEAR.checkValidIntValue(fieldValues.remove(YEAR));
545   if (resolverStyle == ResolverStyle.LENIENT) {
546     long months = Math.subtractExact(fieldValues.remove(MONTH_OF_YEAR), 1);
547     long days = Math.subtractExact(fieldValues.remove(DAY_OF_MONTH), 1);
548     return LocalDate.of(y, 1, 1).plusMonths(months).plusDays(days);
549   }
550   int moy = MONTH_OF_YEAR.checkValidIntValue(fieldValues.remove(MONTH_OF_YEAR));
551   int dom = DAY_OF_MONTH.checkValidIntValue(fieldValues.remove(DAY_OF_MONTH));
552   if (resolverStyle == ResolverStyle.SMART) { // previous valid
553     if (moy == 4 || moy == 6 || moy == 9 || moy == 11) {
554       dom = Math.min(dom, 30);
555     } else if (moy == 2) {
556       dom = Math.min(dom, Month.FEBRUARY.length(Year.isLeap(y)));
557     }
558   }
559   return LocalDate.of(y, moy, dom);
560 }
561
562 //-----
563 @Override
564 public ValueRange range(ChronoField field) {
565   return field.range();
566 }
567
568 //-----
569 /**
570  * Obtains a period for this chronology based on years, months and days.
571  * <p>
572  * This returns a period tied to the ISO chronology using the specified
573  * years, months and days. See {@link Period} for further details.
574  *
575  * @param years the number of years, may be negative
576  * @param months the number of years, may be negative
577  * @param days the number of years, may be negative
578  * @return the period in terms of this chronology, not null
579  * @return the ISO period, not null
580  */
581 @Override // override with covariant return type
582 public Period period(int years, int months, int days) {
583   return Period.of(years, months, days);
584 }
585
586 //-----
587

```

```

588  /**
589   * Writes the Chronology using a
590   * <a href="../../serialized-form.html#java.time.chrono.Ser">dedicated serialized form</a>.
591   * @serialData
592   * <pre>
593   *   out.writeByte(1);    // identifies a Chronology
594   *   out.writeUTF(getId());
595   * </pre>
596   *
597   * @return the instance of {@code Ser}, not null
598   */
599  @Override
600  Object writeReplace() {
601      return super.writeReplace();
602  }
603
604  /**
605   * Defend against malicious streams.
606   *
607   * @param s the stream to read
608   * @throws InvalidObjectException always
609   */
610  private void readObject(ObjectInputStream s) throws InvalidObjectException {
611      throw new InvalidObjectException("Deserialization via serialization delegate");
612  }
613 }

```

5.16 IsoEra.java

Secuencia 86: java/time/chrono/IsoEra.java

```

1  /*
2   * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3   * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4   *
5   *
6   *
7   *
8   *
9   *
10  *
11  *
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *

```

```

24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.chrono;
63
64  import java.time.DateTimeException;
65
66  /**
67   * An era in the ISO calendar system.
68   * <p>
69   * The ISO-8601 standard does not define eras.
70   * A definition has therefore been created with two eras - 'Current era' (CE) for
71   * years on or after 0001-01-01 (ISO), and 'Before current era' (BCE) for years before that.
72   *
73   * <table summary="ISO years and eras" cellpadding="2" cellspacing="3" border="0" >
74   * <thead>
75   * <tr class="tableSubHeadingColor">
76   * <th class="colFirst" align="left">year-of-era</th>

```

```

77 * <th class="colFirst" align="left">era</th>
78 * <th class="colLast" align="left">proleptic-year</th>
79 * </tr>
80 * </thead>
81 * <tbody>
82 * <tr class="rowColor">
83 * <td>2</td><td>CE</td><td>2</td>
84 * </tr>
85 * <tr class="altColor">
86 * <td>1</td><td>CE</td><td>1</td>
87 * </tr>
88 * <tr class="rowColor">
89 * <td>1</td><td>BCE</td><td>0</td>
90 * </tr>
91 * <tr class="altColor">
92 * <td>2</td><td>BCE</td><td>-1</td>
93 * </tr>
94 * </tbody>
95 * </table>
96 * <p>
97 * <b>Do not use {@code ordinal()} to obtain the numeric representation of {@code IsoEra}.
98 * Use {@code getValue()} instead.</b>
99 *
100 * @implSpec
101 * This is an immutable and thread-safe enum.
102 *
103 * @since 1.8
104 */
105 public enum IsoEra implements Era {
106
107     /**
108      * The singleton instance for the era before the current one, 'Before Current Era',
109      * which has the numeric value 0.
110      */
111     BCE,
112     /**
113      * The singleton instance for the current era, 'Current Era',
114      * which has the numeric value 1.
115      */
116     CE;
117
118     //-----
119     /**
120      * Obtains an instance of {@code IsoEra} from an {@code int} value.
121      * <p>
122      * {@code IsoEra} is an enum representing the ISO eras of BCE/CE.
123      * This factory allows the enum to be obtained from the {@code int} value.
124      *
125      * @param isoEra the BCE/CE value to represent, from 0 (BCE) to 1 (CE)
126      * @return the era singleton, not null
127      * @throws DateTimeException if the value is invalid
128      */
129     public static IsoEra of(int isoEra) {

```

```

130     switch (isoEra) {
131         case 0:
132             return BCE;
133         case 1:
134             return CE;
135         default:
136             throw new DateTimeException("Invalid era: " + isoEra);
137     }
138 }
139
140 //-----
141 /**
142  * Gets the numeric era {@code int} value.
143  * <p>
144  * The era BCE has the value 0, while the era CE has the value 1.
145  *
146  * @return the era value, from 0 (BCE) to 1 (CE)
147  */
148 @Override
149 public int getValue() {
150     return ordinal();
151 }
152
153 }

```

5.17 JapaneseChronology.java

Secuencia 87: java/time/chrono/JapaneseChronology.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25

```

```
26  /*
27  * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
28  *
29  * All rights reserved.
30  *
31  * Redistribution and use in source and binary forms, with or without
32  * modification, are permitted provided that the following conditions are met:
33  *
34  * * Redistributions of source code must retain the above copyright notice,
35  *   this list of conditions and the following disclaimer.
36  *
37  * * Redistributions in binary form must reproduce the above copyright notice,
38  *   this list of conditions and the following disclaimer in the documentation
39  *   and/or other materials provided with the distribution.
40  *
41  * * Neither the name of JSR-310 nor the names of its contributors
42  *   may be used to endorse or promote products derived from this software
43  *   without specific prior written permission.
44  *
45  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56  */
57  package java.time.chrono;
58
59  import static java.time.temporal.ChronoField.DAY_OF_MONTH;
60  import static java.time.temporal.ChronoField.DAY_OF_YEAR;
61  import static java.time.temporal.ChronoField.ERA;
62  import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
63  import static java.time.temporal.ChronoField.YEAR;
64  import static java.time.temporal.ChronoField.YEAR_OF_ERA;
65  import static java.time.temporal.ChronoUnit.DAYS;
66  import static java.time.temporal.ChronoUnit.MONTHS;
67
68  import java.io.InvalidObjectException;
69  import java.io.ObjectInputStream;
70  import java.io.Serializable;
71  import java.time.Clock;
72  import java.time.DateTimeException;
73  import java.time.Instant;
74  import java.time.LocalDate;
75  import java.time.Year;
76  import java.time.ZoneId;
77  import java.time.format.ResolverStyle;
78  import java.time.temporal.ChronoField;
```

```

79 import java.time.temporal.TemporalAccessor;
80 import java.time.temporal.TemporalAdjusters;
81 import java.time.temporal.TemporalField;
82 import java.time.temporal.UnsupportedTemporalTypeException;
83 import java.time.temporal.ValueRange;
84 import java.util.Arrays;
85 import java.util.Calendar;
86 import java.util.List;
87 import java.util.Locale;
88 import java.util.Map;
89
90 import sun.util.calendar.CalendarSystem;
91 import sun.util.calendar.LocalGregorianCalendar;
92
93 /**
94  * The Japanese Imperial calendar system.
95  * <p>
96  * This chronology defines the rules of the Japanese Imperial calendar system.
97  * This calendar system is primarily used in Japan.
98  * The Japanese Imperial calendar system is the same as the ISO calendar system
99  * apart from the era-based year numbering.
100 * <p>
101 * Japan introduced the Gregorian calendar starting with Meiji 6.
102 * Only Meiji and later eras are supported;
103 * dates before Meiji 6, January 1 are not supported.
104 * <p>
105 * The supported {@code ChronoField} instances are:
106 * <ul>
107 * <li>{@code DAY_OF_WEEK}
108 * <li>{@code DAY_OF_MONTH}
109 * <li>{@code DAY_OF_YEAR}
110 * <li>{@code EPOCH_DAY}
111 * <li>{@code MONTH_OF_YEAR}
112 * <li>{@code PROLEPTIC_MONTH}
113 * <li>{@code YEAR_OF_ERA}
114 * <li>{@code YEAR}
115 * <li>{@code ERA}
116 * </ul>
117 *
118 * @implSpec
119 * This class is immutable and thread-safe.
120 *
121 * @since 1.8
122 */
123 public final class JapaneseChronology extends AbstractChronology implements Serializable {
124
125     static final LocalGregorianCalendar JCAL =
126         (LocalGregorianCalendar) CalendarSystem.forName("japanese");
127
128     // Locale for creating a JapaneseImperialCalendar.
129     static final Locale LOCALE = Locale.forLanguageTag("ja-JP-u-ca-japanese");
130
131     /**

```

```

132     * Singleton instance for Japanese chronology.
133     */
134     public static final JapaneseChronology INSTANCE = new JapaneseChronology();
135
136     /**
137     * Serialization version.
138     */
139     private static final long serialVersionUID = 459996390165777884L;
140
141     //-----
142     /**
143     * Restricted constructor.
144     */
145     private JapaneseChronology() {
146     }
147
148     //-----
149     /**
150     * Gets the ID of the chronology - 'Japanese'.
151     * <p>
152     * The ID uniquely identifies the {@code Chronology}.
153     * It can be used to lookup the {@code Chronology} using {@link Chronology#of(String)}.
154     *
155     * @return the chronology ID - 'Japanese'
156     * @see #getCalendarType()
157     */
158     @Override
159     public String getId() {
160         return "Japanese";
161     }
162
163     /**
164     * Gets the calendar type of the underlying calendar system - 'japanese'.
165     * <p>
166     * The calendar type is an identifier defined by the
167     * <em>Unicode Locale Data Markup Language (LDML)</em> specification.
168     * It can be used to lookup the {@code Chronology} using {@link Chronology#of(String)}.
169     * It can also be used as part of a locale, accessible via
170     * {@link Locale#getUnicodeLocaleType(String)} with the key 'ca'.
171     *
172     * @return the calendar system type - 'japanese'
173     * @see #getId()
174     */
175     @Override
176     public String getCalendarType() {
177         return "japanese";
178     }
179
180     //-----
181     /**
182     * Obtains a local date in Japanese calendar system from the
183     * era, year-of-era, month-of-year and day-of-month fields.
184     * <p>

```



```

185     * The Japanese month and day-of-month are the same as those in the
186     * ISO calendar system. They are not reset when the era changes.
187     * For example:
188     * <pre>
189     *   6th Jan Showa 64 = ISO 1989-01-06
190     *   7th Jan Showa 64 = ISO 1989-01-07
191     *   8th Jan Heisei 1 = ISO 1989-01-08
192     *   9th Jan Heisei 1 = ISO 1989-01-09
193     * </pre>
194     *
195     * @param era    the Japanese era, not null
196     * @param yearOfEra  the year-of-era
197     * @param month    the month-of-year
198     * @param dayOfMonth  the day-of-month
199     * @return the Japanese local date, not null
200     * @throws DateTimeException if unable to create the date
201     * @throws ClassCastException if the {@code era} is not a {@code JapaneseEra}
202     */
203     @Override
204     public JapaneseDate date(Era era, int yearOfEra, int month, int dayOfMonth) {
205         if (era instanceof JapaneseEra == false) {
206             throw new ClassCastException("Era must be JapaneseEra");
207         }
208         return JapaneseDate.of((JapaneseEra) era, yearOfEra, month, dayOfMonth);
209     }
210
211     /**
212     * Obtains a local date in Japanese calendar system from the
213     * proleptic-year, month-of-year and day-of-month fields.
214     * <p>
215     * The Japanese proleptic year, month and day-of-month are the same as those
216     * in the ISO calendar system. They are not reset when the era changes.
217     *
218     * @param prolepticYear  the proleptic-year
219     * @param month          the month-of-year
220     * @param dayOfMonth     the day-of-month
221     * @return the Japanese local date, not null
222     * @throws DateTimeException if unable to create the date
223     */
224     @Override
225     public JapaneseDate date(int prolepticYear, int month, int dayOfMonth) {
226         return new JapaneseDate(LocalDate.of(prolepticYear, month, dayOfMonth));
227     }
228
229     /**
230     * Obtains a local date in Japanese calendar system from the
231     * era, year-of-era and day-of-year fields.
232     * <p>
233     * The day-of-year in this factory is expressed relative to the start of the year-of-era.
234     * This definition changes the normal meaning of day-of-year only in those years
235     * where the year-of-era is reset to one due to a change in the era.
236     * For example:
237     * <pre>

```

```

238 * 6th Jan Showa 64 = day-of-year 6
239 * 7th Jan Showa 64 = day-of-year 7
240 * 8th Jan Heisei 1 = day-of-year 1
241 * 9th Jan Heisei 1 = day-of-year 2
242 * </pre>
243 *
244 * @param era the Japanese era, not null
245 * @param yearOfEra the year-of-era
246 * @param dayOfYear the day-of-year
247 * @return the Japanese local date, not null
248 * @throws DateTimeException if unable to create the date
249 * @throws ClassCastException if the {@code era} is not a {@code JapaneseEra}
250 */
251 @Override
252 public JapaneseDate dateYearDay(Era era, int yearOfEra, int dayOfYear) {
253     return JapaneseDate.ofYearDay((JapaneseEra) era, yearOfEra, dayOfYear);
254 }
255
256 /**
257 * Obtains a local date in Japanese calendar system from the
258 * proleptic-year and day-of-year fields.
259 * <p>
260 * The day-of-year in this factory is expressed relative to the start of the proleptic year.
261 * The Japanese proleptic year and day-of-year are the same as those in the ISO calendar system
262 * .
263 * They are not reset when the era changes.
264 *
265 * @param prolepticYear the proleptic-year
266 * @param dayOfYear the day-of-year
267 * @return the Japanese local date, not null
268 * @throws DateTimeException if unable to create the date
269 */
270 @Override
271 public JapaneseDate dateYearDay(int prolepticYear, int dayOfYear) {
272     return new JapaneseDate(LocalDate.ofYearDay(prolepticYear, dayOfYear));
273 }
274
275 /**
276 * Obtains a local date in the Japanese calendar system from the epoch-day.
277 *
278 * @param epochDay the epoch day
279 * @return the Japanese local date, not null
280 * @throws DateTimeException if unable to create the date
281 */
282 @Override // override with covariant return type
283 public JapaneseDate dateEpochDay(long epochDay) {
284     return new JapaneseDate(LocalDate.ofEpochDay(epochDay));
285 }
286
287 @Override
288 public JapaneseDate dateNow() {
289     return dateNow(Clock.systemDefaultZone());
290 }

```

```

290
291     @Override
292     public JapaneseDate dateNow(ZoneId zone) {
293         return dateNow(Clock.system(zone));
294     }
295
296     @Override
297     public JapaneseDate dateNow(Clock clock) {
298         return date(LocalDate.now(clock));
299     }
300
301     @Override
302     public JapaneseDate date(TemporalAccessor temporal) {
303         if (temporal instanceof JapaneseDate) {
304             return (JapaneseDate) temporal;
305         }
306         return new JapaneseDate(LocalDate.from(temporal));
307     }
308
309     @Override
310     @SuppressWarnings("unchecked")
311     public ChronoLocalDateTime<JapaneseDate> localDateTime(TemporalAccessor temporal) {
312         return (ChronoLocalDateTime<JapaneseDate>)super.localDateTime(temporal);
313     }
314
315     @Override
316     @SuppressWarnings("unchecked")
317     public ChronoZonedDateTime<JapaneseDate> zonedDateTime(TemporalAccessor temporal) {
318         return (ChronoZonedDateTime<JapaneseDate>)super.zonedDateTime(temporal);
319     }
320
321     @Override
322     @SuppressWarnings("unchecked")
323     public ChronoZonedDateTime<JapaneseDate> zonedDateTime(Instant instant, ZoneId zone) {
324         return (ChronoZonedDateTime<JapaneseDate>)super.zonedDateTime(instant, zone);
325     }
326
327     //-----
328     /**
329      * Checks if the specified year is a leap year.
330      * <p>
331      * Japanese calendar leap years occur exactly in line with ISO leap years.
332      * This method does not validate the year passed in, and only has a
333      * well-defined result for years in the supported range.
334      *
335      * @param prolepticYear the proleptic-year to check, not validated for range
336      * @return true if the year is a leap year
337      */
338     @Override
339     public boolean isLeapYear(long prolepticYear) {
340         return IsoChronology.INSTANCE.isLeapYear(prolepticYear);
341     }
342

```

```

343     @Override
344     public int prolepticYear(Era era, int yearOfEra) {
345         if (era instanceof JapaneseEra == false) {
346             throw new ClassCastException("Era must be JapaneseEra");
347         }
348
349         JapaneseEra jera = (JapaneseEra) era;
350         int gregorianYear = jera.getPrivateEra().getSinceDate().getYear() + yearOfEra - 1;
351         if (yearOfEra == 1) {
352             return gregorianYear;
353         }
354         if (gregorianYear >= Year.MIN_VALUE && gregorianYear <= Year.MAX_VALUE) {
355             LocalGregorianCalendar.Date jdate = JCAL.newCalendarDate(null);
356             jdate.setEra(jera.getPrivateEra()).setDate(yearOfEra, 1, 1);
357             if (JapaneseChronology.JCAL.validate(jdate)) {
358                 return gregorianYear;
359             }
360         }
361         throw new DateTimeException("Invalid yearOfEra value");
362     }
363
364     /**
365      * Returns the calendar system era object from the given numeric value.
366      *
367      * See the description of each Era for the numeric values of:
368      * {@link JapaneseEra#HEISEI}, {@link JapaneseEra#SHOWA}, {@link JapaneseEra#TAISHO},
369      * {@link JapaneseEra#MEIJI}), only Meiji and later eras are supported.
370      *
371      * @param eraValue the era value
372      * @return the Japanese {@code Era} for the given numeric era value
373      * @throws DateTimeException if {@code eraValue} is invalid
374      */
375     @Override
376     public JapaneseEra eraOf(int eraValue) {
377         return JapaneseEra.of(eraValue);
378     }
379
380     @Override
381     public List<Era> eras() {
382         return Arrays.<Era>asList(JapaneseEra.values());
383     }
384
385     JapaneseEra getCurrentEra() {
386         // Assume that the last JapaneseEra is the current one.
387         JapaneseEra[] eras = JapaneseEra.values();
388         return eras[eras.length - 1];
389     }
390
391     //-----
392     @Override
393     public ValueRange range(ChronoField field) {
394         switch (field) {
395             case ALIGNED_DAY_OF_WEEK_IN_MONTH:

```

```

396         case ALIGNED_DAY_OF_WEEK_IN_YEAR:
397         case ALIGNED_WEEK_OF_MONTH:
398         case ALIGNED_WEEK_OF_YEAR:
399             throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
400         case YEAR_OF_ERA: {
401             Calendar jcal = Calendar.getInstance(LOCALE);
402             int startYear = getCurrentEra().getPrivateEra().getSinceDate().getYear();
403             return ValueRange.of(1, jcal.getGreatestMinimum(Calendar.YEAR),
404                 jcal.getLeastMaximum(Calendar.YEAR) + 1, // +1 due to the different
405                     definitions
406                     Year.MAX_VALUE - startYear);
407         }
408         case DAY_OF_YEAR: {
409             Calendar jcal = Calendar.getInstance(LOCALE);
410             int fieldIndex = Calendar.DAY_OF_YEAR;
411             return ValueRange.of(jcal.getMinimum(fieldIndex), jcal.getGreatestMinimum(
412                 fieldIndex),
413                 jcal.getLeastMaximum(fieldIndex), jcal.getMaximum(fieldIndex));
414         }
415         case YEAR:
416             return ValueRange.of(JapaneseDate.MEIJI_6_ISODATE.getYear(), Year.MAX_VALUE);
417         case ERA:
418             return ValueRange.of(JapaneseEra.MEIJI.getValue(), getCurrentEra().getValue());
419         default:
420             return field.range();
421     }
422 }
423
424 //-----
425 @Override // override for return type
426 public JapaneseDate resolveDate(Map <TemporalField, Long> fieldValues, ResolverStyle
427     resolverStyle) {
428     return (JapaneseDate) super.resolveDate(fieldValues, resolverStyle);
429 }
430
431 @Override // override for special Japanese behavior
432 ChronoLocalDate resolveYearOfEra(Map<TemporalField, Long> fieldValues, ResolverStyle
433     resolverStyle) {
434     // validate era and year-of-era
435     Long eraLong = fieldValues.get(ERA);
436     JapaneseEra era = null;
437     if (eraLong != null) {
438         era = eraOf(range(ERA).checkValidIntValue(eraLong, ERA)); // always validated
439     }
440     Long yoeLong = fieldValues.get(YEAR_OF_ERA);
441     int yoe = 0;
442     if (yoeLong != null) {
443         yoe = range(YEAR_OF_ERA).checkValidIntValue(yoeLong, YEAR_OF_ERA); // always validated
444     }
445     // if only year-of-era and no year then invent era unless strict
446     if (era == null && yoeLong != null && fieldValues.containsKey(YEAR) == false &&
447         resolverStyle != ResolverStyle.STRICT) {
448         era = JapaneseEra.values()[JapaneseEra.values().length - 1];
449     }
450 }

```

```

444     }
445     // if both present, then try to create date
446     if (yoeLong != null && era != null) {
447         if (fieldValues.containsKey(MONTH_OF_YEAR)) {
448             if (fieldValues.containsKey(DAY_OF_MONTH)) {
449                 return resolveYMD(era, yoe, fieldValues, resolverStyle);
450             }
451         }
452         if (fieldValues.containsKey(DAY_OF_YEAR)) {
453             return resolveYD(era, yoe, fieldValues, resolverStyle);
454         }
455     }
456     return null;
457 }
458
459 private int prolepticYearLenient(JapaneseEra era, int yearOfEra) {
460     return era.getPrivateEra().getSinceDate().getYear() + yearOfEra - 1;
461 }
462
463 private ChronoLocalDate resolveYMD(JapaneseEra era, int yoe, Map<TemporalField, Long>
    fieldValues, ResolverStyle resolverStyle) {
464     fieldValues.remove(ERA);
465     fieldValues.remove(YEAR_OF_ERA);
466     if (resolverStyle == ResolverStyle.LENIENT) {
467         int y = prolepticYearLenient(era, yoe);
468         long months = Math.subtractExact(fieldValues.remove(MONTH_OF_YEAR), 1);
469         long days = Math.subtractExact(fieldValues.remove(DAY_OF_MONTH), 1);
470         return date(y, 1, 1).plus(months, MONTHS).plus(days, DAYS);
471     }
472     int moy = range(MONTH_OF_YEAR).checkValidIntValue(fieldValues.remove(MONTH_OF_YEAR),
        MONTH_OF_YEAR);
473     int dom = range(DAY_OF_MONTH).checkValidIntValue(fieldValues.remove(DAY_OF_MONTH),
        DAY_OF_MONTH);
474     if (resolverStyle == ResolverStyle.SMART) { // previous valid
475         if (yoe < 1) {
476             throw new DateTimeException("Invalid YearOfEra: " + yoe);
477         }
478         int y = prolepticYearLenient(era, yoe);
479         JapaneseDate result;
480         try {
481             result = date(y, moy, dom);
482         } catch (DateTimeException ex) {
483             result = date(y, moy, 1).with(TemporalAdjusters.lastDayOfMonth());
484         }
485         // handle the era being changed
486         // only allow if the new date is in the same Jan-Dec as the era change
487         // determine by ensuring either original yoe or result yoe is 1
488         if (result.getEra() != era && result.get(YEAR_OF_ERA) > 1 && yoe > 1) {
489             throw new DateTimeException("Invalid YearOfEra for Era: " + era + " " + yoe);
490         }
491         return result;
492     }
493     return date(era, yoe, moy, dom);

```

```

494     }
495
496     private ChronoLocalDate resolveYD(JapaneseEra era, int yoe, Map <TemporalField,Long>
        fieldValues, ResolverStyle resolverStyle) {
497         fieldValues.remove(ERA);
498         fieldValues.remove(YEAR_OF_ERA);
499         if (resolverStyle == ResolverStyle.LENIENT) {
500             int y = prolepticYearLenient(era, yoe);
501             long days = Math.subtractExact(fieldValues.remove(DAY_OF_YEAR), 1);
502             return dateYearDay(y, 1).plus(days, DAYS);
503         }
504         int doy = range(DAY_OF_YEAR).checkValidIntValue(fieldValues.remove(DAY_OF_YEAR),
            DAY_OF_YEAR);
505         return dateYearDay(era, yoe, doy); // smart is same as strict
506     }
507
508     //-----
509     /**
510      * Writes the Chronology using a
511      * <a href="../../../../serialized-form.html#java.time.chrono.Ser">dedicated serialized form</a>.
512      * @serialData
513      * <pre>
514      *   out.writeByte(1);    // identifies a Chronology
515      *   out.writeUTF(getId());
516      * </pre>
517      *
518      * @return the instance of {@code Ser}, not null
519      */
520     @Override
521     Object writeReplace() {
522         return super.writeReplace();
523     }
524
525     /**
526      * Defend against malicious streams.
527      *
528      * @param s the stream to read
529      * @throws InvalidObjectException always
530      */
531     private void readObject(ObjectInputStream s) throws InvalidObjectException {
532         throw new InvalidObjectException("Deserialization via serialization delegate");
533     }
534 }

```

5.18 JapaneseDate.java

Secuencia 88: java/time/chrono/JapaneseDate.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *

```

```
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27  * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
28  *
29  * All rights reserved.
30  *
31  * Redistribution and use in source and binary forms, with or without
32  * modification, are permitted provided that the following conditions are met:
33  *
34  * * Redistributions of source code must retain the above copyright notice,
35  *   this list of conditions and the following disclaimer.
36  *
37  * * Redistributions in binary form must reproduce the above copyright notice,
38  *   this list of conditions and the following disclaimer in the documentation
39  *   and/or other materials provided with the distribution.
40  *
41  * * Neither the name of JSR-310 nor the names of its contributors
42  *   may be used to endorse or promote products derived from this software
43  *   without specific prior written permission.
44  *
45  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56  */
57 package java.time.chrono;
58
59 import static java.time.temporal.ChronoField.ALIGNED_DAY_OF_WEEK_IN_MONTH;
```



```
60 import static java.time.temporal.ChronoField.ALIGNED_DAY_OF_WEEK_IN_YEAR;
61 import static java.time.temporal.ChronoField.ALIGNED_WEEK_OF_MONTH;
62 import static java.time.temporal.ChronoField.ALIGNED_WEEK_OF_YEAR;
63 import static java.time.temporal.ChronoField.DAY_OF_MONTH;
64 import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
65 import static java.time.temporal.ChronoField.YEAR;
66
67 import java.io.DataInput;
68 import java.io.DataOutput;
69 import java.io.IOException;
70 import java.io.InvalidObjectException;
71 import java.io.ObjectInputStream;
72 import java.io.Serializable;
73 import java.time.Clock;
74 import java.time.DateTimeException;
75 import java.time.LocalDate;
76 import java.time.LocalDateTime;
77 import java.time.Period;
78 import java.time.ZoneId;
79 import java.time.temporal.ChronoField;
80 import java.time.temporal.TemporalAccessor;
81 import java.time.temporal.TemporalAdjuster;
82 import java.time.temporal.TemporalAmount;
83 import java.time.temporal.TemporalField;
84 import java.time.temporal.TemporalQuery;
85 import java.time.temporal.TemporalUnit;
86 import java.time.temporal.UnsupportedTemporalTypeException;
87 import java.time.temporal.ValueRange;
88 import java.util.Calendar;
89 import java.util.Objects;
90
91 import sun.util.calendar.CalendarDate;
92 import sun.util.calendar.LocalGregorianCalendar;
93
94 /**
95  * A date in the Japanese Imperial calendar system.
96  * <p>
97  * This date operates using the {@linkplain JapaneseChronology Japanese Imperial calendar}.
98  * This calendar system is primarily used in Japan.
99  * <p>
100 * The Japanese Imperial calendar system is the same as the ISO calendar system
101 * apart from the era-based year numbering. The proleptic-year is defined to be
102 * equal to the ISO proleptic-year.
103 * <p>
104 * Japan introduced the Gregorian calendar starting with Meiji 6.
105 * Only Meiji and later eras are supported;
106 * dates before Meiji 6, January 1 are not supported.
107 * <p>
108 * For example, the Japanese year "Heisei 24" corresponds to ISO year "2012".<br>
109 * Calling {@code japaneseDate.get(YEAR_OF_ERA)} will return 24.<br>
110 * Calling {@code japaneseDate.get(YEAR)} will return 2012.<br>
111 * Calling {@code japaneseDate.get(ERA)} will return 2, corresponding to
112 * {@code JapaneseChronology.ERA_HEISEI}.<br>
```

```

113  *
114  * <p>
115  * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
116  * class; use of identity-sensitive operations (including reference equality
117  * ({@code ==}), identity hash code, or synchronization) on instances of
118  * {@code JapaneseDate} may have unpredictable results and should be avoided.
119  * The {@code equals} method should be used for comparisons.
120  *
121  * @implSpec
122  * This class is immutable and thread-safe.
123  *
124  * @since 1.8
125  */
126 public final class JapaneseDate
127     extends ChronoLocalDateImpl<JapaneseDate>
128     implements ChronoLocalDate, Serializable {
129
130     /**
131      * Serialization version.
132      */
133     private static final long serialVersionUID = -305327627230580483L;
134
135     /**
136      * The underlying ISO local date.
137      */
138     private final transient LocalDate isoDate;
139     /**
140      * The JapaneseEra of this date.
141      */
142     private transient JapaneseEra era;
143     /**
144      * The Japanese imperial calendar year of this date.
145      */
146     private transient int yearOfEra;
147
148     /**
149      * The first day supported by the JapaneseChronology is Meiji 6, January 1st.
150      */
151     static final LocalDate MEIJI_6_ISODATE = LocalDate.of(1873, 1, 1);
152
153     //-----
154     /**
155      * Obtains the current {@code JapaneseDate} from the system clock in the default time-zone.
156      * <p>
157      * This will query the {@link Clock#systemDefaultZone() system clock} in the default
158      * time-zone to obtain the current date.
159      * <p>
160      * Using this method will prevent the ability to use an alternate clock for testing
161      * because the clock is hard-coded.
162      *
163      * @return the current date using the system clock and default time-zone, not null
164      */
165     public static JapaneseDate now() {

```

```

166         return now(Clock.systemDefaultZone());
167     }
168
169     /**
170      * Obtains the current {@code JapaneseDate} from the system clock in the specified time-zone.
171      * <p>
172      * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current date.
173      * Specifying the time-zone avoids dependence on the default time-zone.
174      * <p>
175      * Using this method will prevent the ability to use an alternate clock for testing
176      * because the clock is hard-coded.
177      *
178      * @param zone the zone ID to use, not null
179      * @return the current date using the system clock, not null
180      */
181     public static JapaneseDate now(ZoneId zone) {
182         return now(Clock.system(zone));
183     }
184
185     /**
186      * Obtains the current {@code JapaneseDate} from the specified clock.
187      * <p>
188      * This will query the specified clock to obtain the current date - today.
189      * Using this method allows the use of an alternate clock for testing.
190      * The alternate clock may be introduced using {@linkplain Clock dependency injection}.
191      *
192      * @param clock the clock to use, not null
193      * @return the current date, not null
194      * @throws DateTimeException if the current date cannot be obtained
195      */
196     public static JapaneseDate now(Clock clock) {
197         return new JapaneseDate(LocalDate.now(clock));
198     }
199
200     /**
201      * Obtains a {@code JapaneseDate} representing a date in the Japanese calendar
202      * system from the era, year-of-era, month-of-year and day-of-month fields.
203      * <p>
204      * This returns a {@code JapaneseDate} with the specified fields.
205      * The day must be valid for the year and month, otherwise an exception will be thrown.
206      * <p>
207      * The Japanese month and day-of-month are the same as those in the
208      * ISO calendar system. They are not reset when the era changes.
209      * For example:
210      * <pre>
211      * 6th Jan Showa 64 = ISO 1989-01-06
212      * 7th Jan Showa 64 = ISO 1989-01-07
213      * 8th Jan Heisei 1 = ISO 1989-01-08
214      * 9th Jan Heisei 1 = ISO 1989-01-09
215      * </pre>
216      *
217      * @param era the Japanese era, not null
218      * @param yearOfEra the Japanese year-of-era

```

```

219 * @param month the Japanese month-of-year, from 1 to 12
220 * @param dayOfMonth the Japanese day-of-month, from 1 to 31
221 * @return the date in Japanese calendar system, not null
222 * @throws DateTimeException if the value of any field is out of range,
223 * or if the day-of-month is invalid for the month-year,
224 * or if the date is not a Japanese era
225 */
226 public static JapaneseDate of(JapaneseEra era, int yearOfEra, int month, int dayOfMonth) {
227     Objects.requireNonNull(era, "era");
228     LocalGregorianCalendar.Date jdate = JapaneseChronology.JCAL.newCalendarDate(null);
229     jdate.setEra(era.getPrivateEra()).setDate(yearOfEra, month, dayOfMonth);
230     if (!JapaneseChronology.JCAL.validate(jdate)) {
231         throw new DateTimeException("year, month, and day not valid for Era");
232     }
233     LocalDate date = LocalDate.of(jdate.getNormalizedYear(), month, dayOfMonth);
234     return new JapaneseDate(era, yearOfEra, date);
235 }
236
237 /**
238  * Obtains a {@code JapaneseDate} representing a date in the Japanese calendar
239  * system from the proleptic-year, month-of-year and day-of-month fields.
240  * <p>
241  * This returns a {@code JapaneseDate} with the specified fields.
242  * The day must be valid for the year and month, otherwise an exception will be thrown.
243  * <p>
244  * The Japanese proleptic year, month and day-of-month are the same as those
245  * in the ISO calendar system. They are not reset when the era changes.
246  *
247  * @param prolepticYear the Japanese proleptic-year
248  * @param month the Japanese month-of-year, from 1 to 12
249  * @param dayOfMonth the Japanese day-of-month, from 1 to 31
250  * @return the date in Japanese calendar system, not null
251  * @throws DateTimeException if the value of any field is out of range,
252  * or if the day-of-month is invalid for the month-year
253  */
254 public static JapaneseDate of(int prolepticYear, int month, int dayOfMonth) {
255     return new JapaneseDate(LocalDate.of(prolepticYear, month, dayOfMonth));
256 }
257
258 /**
259  * Obtains a {@code JapaneseDate} representing a date in the Japanese calendar
260  * system from the era, year-of-era and day-of-year fields.
261  * <p>
262  * This returns a {@code JapaneseDate} with the specified fields.
263  * The day must be valid for the year, otherwise an exception will be thrown.
264  * <p>
265  * The day-of-year in this factory is expressed relative to the start of the year-of-era.
266  * This definition changes the normal meaning of day-of-year only in those years
267  * where the year-of-era is reset to one due to a change in the era.
268  * For example:
269  * <pre>
270  * 6th Jan Showa 64 = day-of-year 6
271  * 7th Jan Showa 64 = day-of-year 7

```

```

272 * 8th Jan Heisei 1 = day-of-year 1
273 * 9th Jan Heisei 1 = day-of-year 2
274 * </pre>
275 *
276 * @param era the Japanese era, not null
277 * @param yearOfEra the Japanese year-of-era
278 * @param dayOfYear the chronology day-of-year, from 1 to 366
279 * @return the date in Japanese calendar system, not null
280 * @throws DateTimeException if the value of any field is out of range,
281 * or if the day-of-year is invalid for the year
282 */
283 static JapaneseDate ofYearDay(JapaneseEra era, int yearOfEra, int dayOfYear) {
284     Objects.requireNonNull(era, "era");
285     CalendarDate firstDay = era.getPrivateEra().getSinceDate();
286     LocalGregorianCalendar.Date jdate = JapaneseChronology.JCAL.newCalendarDate(null);
287     jdate.setEra(era.getPrivateEra());
288     if (yearOfEra == 1) {
289         jdate.setDate(yearOfEra, firstDay.getMonth(), firstDay.getDayOfMonth() + dayOfYear - 1)
290     } else {
291         jdate.setDate(yearOfEra, 1, dayOfYear);
292     }
293     JapaneseChronology.JCAL.normalize(jdate);
294     if (era.getPrivateEra() != jdate.getEra() || yearOfEra != jdate.getYear()) {
295         throw new DateTimeException("Invalid parameters");
296     }
297     LocalDate localdate = LocalDate.of(jdate.getNormalizedYear(),
298         jdate.getMonth(), jdate.getDayOfMonth());
299     return new JapaneseDate(era, yearOfEra, localdate);
300 }
301
302 /**
303 * Obtains a {@code JapaneseDate} from a temporal object.
304 * <p>
305 * This obtains a date in the Japanese calendar system based on the specified temporal.
306 * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
307 * which this factory converts to an instance of {@code JapaneseDate}.
308 * <p>
309 * The conversion typically uses the {@link ChronoField#EPOCH_DAY EPOCH_DAY}
310 * field, which is standardized across calendar systems.
311 * <p>
312 * This method matches the signature of the functional interface {@link TemporalQuery}
313 * allowing it to be used as a query via method reference, {@code JapaneseDate::from}.
314 *
315 * @param temporal the temporal object to convert, not null
316 * @return the date in Japanese calendar system, not null
317 * @throws DateTimeException if unable to convert to a {@code JapaneseDate}
318 */
319 public static JapaneseDate from(TemporalAccessor temporal) {
320     return JapaneseChronology.INSTANCE.date(temporal);
321 }
322
323 //-----

```

```

324  /**
325   * Creates an instance from an ISO date.
326   *
327   * @param isoDate the standard local date, validated not null
328   */
329  JapaneseDate(LocalDate isoDate) {
330      if (isoDate.isBefore(MEIJII_6_ISODATE)) {
331          throw new DateTimeException("JapaneseDate before Meiji 6 is not supported");
332      }
333      LocalGregorianCalendar.Date jdate = toPrivateJapaneseDate(isoDate);
334      this.era = JapaneseEra.toJapaneseEra(jdate.getEra());
335      this.yearOfEra = jdate.getYear();
336      this.isoDate = isoDate;
337  }
338
339  /**
340   * Constructs a {@code JapaneseDate}. This constructor does NOT validate the given parameters,
341   * and {@code era} and {@code year} must agree with {@code isoDate}.
342   *
343   * @param era the era, validated not null
344   * @param year the year-of-era, validated
345   * @param isoDate the standard local date, validated not null
346   */
347  JapaneseDate(JapaneseEra era, int year, LocalDate isoDate) {
348      if (isoDate.isBefore(MEIJII_6_ISODATE)) {
349          throw new DateTimeException("JapaneseDate before Meiji 6 is not supported");
350      }
351      this.era = era;
352      this.yearOfEra = year;
353      this.isoDate = isoDate;
354  }
355
356  //-----
357  /**
358   * Gets the chronology of this date, which is the Japanese calendar system.
359   * <p>
360   * The {@code Chronology} represents the calendar system in use.
361   * The era and other fields in {@link ChronoField} are defined by the chronology.
362   *
363   * @return the Japanese chronology, not null
364   */
365  @Override
366  public JapaneseChronology getChronology() {
367      return JapaneseChronology.INSTANCE;
368  }
369
370  /**
371   * Gets the era applicable at this date.
372   * <p>
373   * The Japanese calendar system has multiple eras defined by {@link JapaneseEra}.
374   *
375   * @return the era applicable at this date, not null
376   */

```

```

377     @Override
378     public JapaneseEra getEra() {
379         return era;
380     }
381
382     /**
383      * Returns the length of the month represented by this date.
384      * <p>
385      * This returns the length of the month in days.
386      * Month lengths match those of the ISO calendar system.
387      *
388      * @return the length of the month in days
389      */
390     @Override
391     public int lengthOfMonth() {
392         return isoDate.lengthOfMonth();
393     }
394
395     @Override
396     public int lengthOfYear() {
397         Calendar jcal = Calendar.getInstance(JapaneseChronology.LOCALE);
398         jcal.set(Calendar.ERA, era.getValue() + JapaneseEra.ERA_OFFSET);
399         jcal.set(yearOfEra, isoDate.getMonthValue() - 1, isoDate.getDayOfMonth());
400         return jcal.getActualMaximum(Calendar.DAY_OF_YEAR);
401     }
402
403     //-----
404     /**
405      * Checks if the specified field is supported.
406      * <p>
407      * This checks if this date can be queried for the specified field.
408      * If false, then calling the {@link #range(TemporalField) range} and
409      * {@link #get(TemporalField) get} methods will throw an exception.
410      * <p>
411      * If the field is a {@link ChronoField} then the query is implemented here.
412      * The supported fields are:
413      * <ul>
414      * <li>{@code DAY_OF_WEEK}
415      * <li>{@code DAY_OF_MONTH}
416      * <li>{@code DAY_OF_YEAR}
417      * <li>{@code EPOCH_DAY}
418      * <li>{@code MONTH_OF_YEAR}
419      * <li>{@code PROLEPTIC_MONTH}
420      * <li>{@code YEAR_OF_ERA}
421      * <li>{@code YEAR}
422      * <li>{@code ERA}
423      * </ul>
424      * All other {@code ChronoField} instances will return false.
425      * <p>
426      * If the field is not a {@code ChronoField}, then the result of this method
427      * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
428      * passing {@code this} as the argument.
429      * Whether the field is supported is determined by the field.

```

```

430     *
431     * @param field the field to check, null returns false
432     * @return true if the field is supported on this date, false if not
433     */
434     @Override
435     public boolean isSupported(TemporalField field) {
436         if (field == ALIGNED_DAY_OF_WEEK_IN_MONTH || field == ALIGNED_DAY_OF_WEEK_IN_YEAR ||
437             field == ALIGNED_WEEK_OF_MONTH || field == ALIGNED_WEEK_OF_YEAR) {
438             return false;
439         }
440         return ChronoLocalDate.super.isSupported(field);
441     }
442
443     @Override
444     public ValueRange range(TemporalField field) {
445         if (field instanceof ChronoField) {
446             if (isSupported(field)) {
447                 ChronoField f = (ChronoField) field;
448                 switch (f) {
449                     case DAY_OF_MONTH: return ValueRange.of(1, lengthOfMonth());
450                     case DAY_OF_YEAR: return ValueRange.of(1, lengthOfYear());
451                     case YEAR_OF_ERA: {
452                         Calendar jcal = Calendar.getInstance(JapaneseChronology.LOCALE);
453                         jcal.set(Calendar.ERA, era.getValue() + JapaneseEra.ERA_OFFSET);
454                         jcal.set(yearOfEra, isoDate.getMonthValue() - 1, isoDate.getDayOfMonth());
455                         return ValueRange.of(1, jcal.getActualMaximum(Calendar.YEAR));
456                     }
457                 }
458                 return getChronology().range(f);
459             }
460             throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
461         }
462         return field.rangeRefinedBy(this);
463     }
464
465     @Override
466     public long getLong(TemporalField field) {
467         if (field instanceof ChronoField) {
468             // same as ISO:
469             // DAY_OF_WEEK, DAY_OF_MONTH, EPOCH_DAY, MONTH_OF_YEAR, PROLEPTIC_MONTH, YEAR
470             //
471             // calendar specific fields
472             // DAY_OF_YEAR, YEAR_OF_ERA, ERA
473             switch ((ChronoField) field) {
474                 case ALIGNED_DAY_OF_WEEK_IN_MONTH:
475                 case ALIGNED_DAY_OF_WEEK_IN_YEAR:
476                 case ALIGNED_WEEK_OF_MONTH:
477                 case ALIGNED_WEEK_OF_YEAR:
478                     throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
479                 case YEAR_OF_ERA:
480                     return yearOfEra;
481                 case ERA:
482                     return era.getValue();

```



```

483         case DAY_OF_YEAR:
484             Calendar jcal = Calendar.getInstance(JapaneseChronology.LOCALE);
485             jcal.set(Calendar.ERA, era.getValue() + JapaneseEra.ERA_OFFSET);
486             jcal.set(yearOfEra, isoDate.getMonthValue() - 1, isoDate.getDayOfMonth());
487             return jcal.get(Calendar.DAY_OF_YEAR);
488         }
489         return isoDate.getLong(field);
490     }
491     return field.getFrom(this);
492 }
493
494 /**
495  * Returns a {@code LocalGregorianCalendar.Date} converted from the given {@code isoDate}.
496  *
497  * @param isoDate the local date, not null
498  * @return a {@code LocalGregorianCalendar.Date}, not null
499  */
500 private static LocalGregorianCalendar.Date toPrivateJapaneseDate(LocalDate isoDate) {
501     LocalGregorianCalendar.Date jdate = JapaneseChronology.JCAL.newCalendarDate(null);
502     sun.util.calendar.Era sunEra = JapaneseEra.privateEraFrom(isoDate);
503     int year = isoDate.getYear();
504     if (sunEra != null) {
505         year -= sunEra.getSinceDate().getYear() - 1;
506     }
507     jdate.setEra(sunEra).setYear(year).setMonth(isoDate.getMonthValue()).setDayOfMonth(isoDate.
508         getDayOfMonth());
509     JapaneseChronology.JCAL.normalize(jdate);
510     return jdate;
511 }
512
513 //-----
514 @Override
515 public JapaneseDate with(TemporalField field, long newValue) {
516     if (field instanceof ChronoField) {
517         ChronoField f = (ChronoField) field;
518         if (getLong(f) == newValue) { // getLong() validates for supported fields
519             return this;
520         }
521         switch (f) {
522             case YEAR_OF_ERA:
523             case YEAR:
524             case ERA: {
525                 int nvalue = getChronology().range(f).checkValidIntValue(newValue, f);
526                 switch (f) {
527                     case YEAR_OF_ERA:
528                         return this.withYear(nvalue);
529                     case YEAR:
530                         return with(isoDate.withYear(nvalue));
531                     case ERA: {
532                         return this.withYear(JapaneseEra.of(nvalue), yearOfEra);
533                     }
534                 }
535             }
536         }
537     }
538 }

```

```

535     }
536     // YEAR, PROLEPTIC_MONTH and others are same as ISO
537     return with(isoDate.with(field, newValue));
538 }
539 return super.with(field, newValue);
540 }
541
542 /**
543  * {@inheritDoc}
544  * @throws DateTimeException {@inheritDoc}
545  * @throws ArithmeticException {@inheritDoc}
546  */
547 @Override
548 public JapaneseDate with(TemporalAdjuster adjuster) {
549     return super.with(adjuster);
550 }
551
552 /**
553  * {@inheritDoc}
554  * @throws DateTimeException {@inheritDoc}
555  * @throws ArithmeticException {@inheritDoc}
556  */
557 @Override
558 public JapaneseDate plus(TemporalAmount amount) {
559     return super.plus(amount);
560 }
561
562 /**
563  * {@inheritDoc}
564  * @throws DateTimeException {@inheritDoc}
565  * @throws ArithmeticException {@inheritDoc}
566  */
567 @Override
568 public JapaneseDate minus(TemporalAmount amount) {
569     return super.minus(amount);
570 }
571 //-----
572 /**
573  * Returns a copy of this date with the year altered.
574  * <p>
575  * This method changes the year of the date.
576  * If the month-day is invalid for the year, then the previous valid day
577  * will be selected instead.
578  * <p>
579  * This instance is immutable and unaffected by this method call.
580  *
581  * @param era the era to set in the result, not null
582  * @param yearOfEra the year-of-era to set in the returned date
583  * @return a {@code JapaneseDate} based on this date with the requested year, never null
584  * @throws DateTimeException if {@code year} is invalid
585  */
586 private JapaneseDate withYear(JapaneseEra era, int yearOfEra) {
587     int year = JapaneseChronology.INSTANCE.prolepticYear(era, yearOfEra);

```

```

588         return with(isoDate.withYear(year));
589     }
590
591     /**
592      * Returns a copy of this date with the year-of-era altered.
593      * <p>
594      * This method changes the year-of-era of the date.
595      * If the month-day is invalid for the year, then the previous valid day
596      * will be selected instead.
597      * <p>
598      * This instance is immutable and unaffected by this method call.
599      *
600      * @param year the year to set in the returned date
601      * @return a {@code JapaneseDate} based on this date with the requested year-of-era, never null
602      * @throws DateTimeException if {@code year} is invalid
603      */
604     private JapaneseDate withYear(int year) {
605         return withYear(getEra(), year);
606     }
607
608     //-----
609     @Override
610     JapaneseDate plusYears(long years) {
611         return with(isoDate.plusYears(years));
612     }
613
614     @Override
615     JapaneseDate plusMonths(long months) {
616         return with(isoDate.plusMonths(months));
617     }
618
619     @Override
620     JapaneseDate plusWeeks(long weeksToAdd) {
621         return with(isoDate.plusWeeks(weeksToAdd));
622     }
623
624     @Override
625     JapaneseDate plusDays(long days) {
626         return with(isoDate.plusDays(days));
627     }
628
629     @Override
630     public JapaneseDate plus(long amountToAdd, TemporalUnit unit) {
631         return super.plus(amountToAdd, unit);
632     }
633
634     @Override
635     public JapaneseDate minus(long amountToAdd, TemporalUnit unit) {
636         return super.minus(amountToAdd, unit);
637     }
638
639     @Override
640     JapaneseDate minusYears(long yearsToSubtract) {

```

```

641         return super.minusYears(yearsToSubtract);
642     }
643
644     @Override
645     JapaneseDate minusMonths(long monthsToSubtract) {
646         return super.minusMonths(monthsToSubtract);
647     }
648
649     @Override
650     JapaneseDate minusWeeks(long weeksToSubtract) {
651         return super.minusWeeks(weeksToSubtract);
652     }
653
654     @Override
655     JapaneseDate minusDays(long daysToSubtract) {
656         return super.minusDays(daysToSubtract);
657     }
658
659     private JapaneseDate with(LocalDate newDate) {
660         return (newDate.equals(isoDate) ? this : new JapaneseDate(newDate));
661     }
662
663     @Override          // for javadoc and covariant return type
664     @SuppressWarnings("unchecked")
665     public final ChronoLocalDateTime<JapaneseDate> atTime(LocalTime localTime) {
666         return (ChronoLocalDateTime<JapaneseDate>)super.atTime(localTime);
667     }
668
669     @Override
670     public ChronoPeriod until(ChronoLocalDate endDate) {
671         Period period = isoDate.until(endDate);
672         return getChronology().period(period.getYears(), period.getMonths(), period.getDays());
673     }
674
675     @Override // override for performance
676     public long toEpochDay() {
677         return isoDate.toEpochDay();
678     }
679
680     //-----
681     /**
682      * Compares this date to another date, including the chronology.
683      * <p>
684      * Compares this {@code JapaneseDate} with another ensuring that the date is the same.
685      * <p>
686      * Only objects of type {@code JapaneseDate} are compared, other types return false.
687      * To compare the dates of two {@code TemporalAccessor} instances, including dates
688      * in two different chronologies, use {@link ChronoField#EPOCH_DAY} as a comparator.
689      *
690      * @param obj the object to check, null returns false
691      * @return true if this is equal to the other date
692      */
693     @Override // override for performance

```

```

694 public boolean equals(Object obj) {
695     if (this == obj) {
696         return true;
697     }
698     if (obj instanceof JapaneseDate) {
699         JapaneseDate otherDate = (JapaneseDate) obj;
700         return this.isoDate.equals(otherDate.isoDate);
701     }
702     return false;
703 }
704
705 /**
706  * A hash code for this date.
707  *
708  * @return a suitable hash code based only on the Chronology and the date
709  */
710 @Override // override for performance
711 public int hashCode() {
712     return getChronology().getId().hashCode() ^ isoDate.hashCode();
713 }
714
715 //-----
716 /**
717  * Defend against malicious streams.
718  *
719  * @param s the stream to read
720  * @throws InvalidObjectException always
721  */
722 private void readObject(ObjectInputStream s) throws InvalidObjectException {
723     throw new InvalidObjectException("Deserialization via serialization delegate");
724 }
725
726 /**
727  * Writes the object using a
728  * <a href="../../../../serialized-form.html#java.time.chrono.Ser">dedicated serialized form</a>.
729  * @serialData
730  * <pre>
731  *   out.writeByte(4); // identifies a JapaneseDate
732  *   out.writeInt(get(YEAR));
733  *   out.writeByte(get(MONTH_OF_YEAR));
734  *   out.writeByte(get(DAY_OF_MONTH));
735  * </pre>
736  *
737  * @return the instance of {@code Ser}, not null
738  */
739 private Object writeReplace() {
740     return new Ser(Ser.JAPANESE_DATE_TYPE, this);
741 }
742
743 void writeExternal(DataOutput out) throws IOException {
744     // JapaneseChronology is implicit in the JAPANESE_DATE_TYPE
745     out.writeInt(get(YEAR));
746     out.writeByte(get(MONTH_OF_YEAR));

```

```
747     out.writeByte(get(DAY_OF_MONTH));
748 }
749
750 static JapaneseDate readExternal(DataInput in) throws IOException {
751     int year = in.readInt();
752     int month = in.readByte();
753     int dayOfMonth = in.readByte();
754     return JapaneseChronology.INSTANCE.date(year, month, dayOfMonth);
755 }
756
757 }
```

5.19 JapaneseEra.java

Secuencia 89: java/time/chrono/JapaneseEra.java

```
1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
```

```
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62 package java.time.chrono;
63
64 import static java.time.chrono.JapaneseDate.MEIJI_6_ISODATE;
65 import static java.time.temporal.ChronoField.ERA;
66
67 import java.io.DataInput;
68 import java.io.DataOutput;
69 import java.io.IOException;
70 import java.io.InvalidObjectException;
71 import java.io.ObjectInputStream;
72 import java.io.ObjectStreamException;
73 import java.io.Serializable;
74 import java.time.DateTimeException;
75 import java.time.LocalDate;
76 import java.time.temporal.ChronoField;
77 import java.time.temporal.TemporalField;
78 import java.time.temporal.UnsupportedTemporalTypeException;
79 import java.time.temporal.ValueRange;
80 import java.util.Arrays;
81 import java.util.Objects;
82
83 import sun.util.calendar.CalendarDate;
84
85 /**
86  * An era in the Japanese Imperial calendar system.
87  * <p>
88  * This class defines the valid eras for the Japanese chronology.
89  * Japan introduced the Gregorian calendar starting with Meiji 6.
90  * Only Meiji and later eras are supported;
91  * dates before Meiji 6, January 1 are not supported.
```

```

92  *
93  * @implSpec
94  * This class is immutable and thread-safe.
95  *
96  * @since 1.8
97  */
98  public final class JapaneseEra
99      implements Era, Serializable {
100
101      // The offset value to 0-based index from the era value.
102      // i.e., getValue() + ERA_OFFSET == 0-based index
103      static final int ERA_OFFSET = 2;
104
105      static final sun.util.calendar.Era[] ERA_CONFIG;
106
107      /**
108       * The singleton instance for the 'Meiji' era (1868-01-01 - 1912-07-29)
109       * which has the value -1.
110       */
111      public static final JapaneseEra MEIJI = new JapaneseEra(-1, LocalDate.of(1868, 1, 1));
112      /**
113       * The singleton instance for the 'Taisho' era (1912-07-30 - 1926-12-24)
114       * which has the value 0.
115       */
116      public static final JapaneseEra TAISHO = new JapaneseEra(0, LocalDate.of(1912, 7, 30));
117      /**
118       * The singleton instance for the 'Showa' era (1926-12-25 - 1989-01-07)
119       * which has the value 1.
120       */
121      public static final JapaneseEra SHOWA = new JapaneseEra(1, LocalDate.of(1926, 12, 25));
122      /**
123       * The singleton instance for the 'Heisei' era (1989-01-08 - current)
124       * which has the value 2.
125       */
126      public static final JapaneseEra HEISEI = new JapaneseEra(2, LocalDate.of(1989, 1, 8));
127
128      // the number of defined JapaneseEra constants.
129      // There could be an extra era defined in its configuration.
130      private static final int N_ERA_CONSTANTS = HEISEI.getValue() + ERA_OFFSET;
131
132      /**
133       * Serialization version.
134       */
135      private static final long serialVersionUID = 1466499369062886794L;
136
137      // array for the singleton JapaneseEra instances
138      private static final JapaneseEra[] KNOWN_ERAS;
139
140      static {
141          ERA_CONFIG = JapaneseChronology.JCAL.getEras();
142
143          KNOWN_ERAS = new JapaneseEra[ERA_CONFIG.length];
144          KNOWN_ERAS[0] = MEIJI;

```



```

145     KNOWN_ERAS[1] = TAISHO;
146     KNOWN_ERAS[2] = SHOWA;
147     KNOWN_ERAS[3] = HEISEI;
148     for (int i = N_ERA_CONSTANTS; i < ERA_CONFIG.length; i++) {
149         CalendarDate date = ERA_CONFIG[i].getSinceDate();
150         LocalDate isoDate = LocalDate.of(date.getYear(), date.getMonth(), date.getDayOfMonth());
151         KNOWN_ERAS[i] = new JapaneseEra(i - ERA_OFFSET + 1, isoDate);
152     }
153 };
154
155 /**
156  * The era value.
157  * @serial
158  */
159 private final transient int eraValue;
160
161 // the first day of the era
162 private final transient LocalDate since;
163
164 /**
165  * Creates an instance.
166  *
167  * @param eraValue the era value, validated
168  * @param since the date representing the first date of the era, validated not null
169  */
170 private JapaneseEra(int eraValue, LocalDate since) {
171     this.eraValue = eraValue;
172     this.since = since;
173 }
174
175 //-----
176 /**
177  * Returns the Sun private Era instance corresponding to this {@code JapaneseEra}.
178  *
179  * @return the Sun private Era instance for this {@code JapaneseEra}.
180  */
181 sun.util.calendar.Era getPrivateEra() {
182     return ERA_CONFIG[ordinal(eraValue)];
183 }
184
185 //-----
186 /**
187  * Obtains an instance of {@code JapaneseEra} from an {@code int} value.
188  * <p>
189  * The {@link #SHOWA} era that contains 1970-01-01 (ISO calendar system) has the value 1
190  * Later era is numbered 2 ({@link #HEISEI}). Earlier eras are numbered 0 ({@link #TAISHO}),
191  * -1 ({@link #MEIJI}), only Meiji and later eras are supported.
192  *
193  * @param japaneseEra the era to represent
194  * @return the {@code JapaneseEra} singleton, not null
195  * @throws DateTimeException if the value is invalid
196  */

```

```

197 public static JapaneseEra of(int japaneseEra) {
198     if (japaneseEra < MEIJI.eraValue || japaneseEra + ERA_OFFSET > KNOWN_ERAS.length) {
199         throw new DateTimeException("Invalid era: " + japaneseEra);
200     }
201     return KNOWN_ERAS[ordinal(japaneseEra)];
202 }
203
204 /**
205  * Returns the {@code JapaneseEra} with the name.
206  * <p>
207  * The string must match exactly the name of the era.
208  * (Extraneous whitespace characters are not permitted.)
209  *
210  * @param japaneseEra the japaneseEra name; non-null
211  * @return the {@code JapaneseEra} singleton, never null
212  * @throws IllegalArgumentException if there is not JapaneseEra with the specified name
213  */
214 public static JapaneseEra valueOf(String japaneseEra) {
215     Objects.requireNonNull(japaneseEra, "japaneseEra");
216     for (JapaneseEra era : KNOWN_ERAS) {
217         if (era.getName().equals(japaneseEra)) {
218             return era;
219         }
220     }
221     throw new IllegalArgumentException("japaneseEra is invalid");
222 }
223
224 /**
225  * Returns an array of JapaneseEras.
226  * <p>
227  * This method may be used to iterate over the JapaneseEras as follows:
228  * <pre>
229  * for (JapaneseEra c : JapaneseEra.values())
230  *     System.out.println(c);
231  * </pre>
232  *
233  * @return an array of JapaneseEras
234  */
235 public static JapaneseEra[] values() {
236     return Arrays.copyOf(KNOWN_ERAS, KNOWN_ERAS.length);
237 }
238
239 //-----
240 /**
241  * Obtains an instance of {@code JapaneseEra} from a date.
242  *
243  * @param date the date, not null
244  * @return the Era singleton, never null
245  */
246 static JapaneseEra from(LocalDate date) {
247     if (date.isBefore(MEIJI_6_ISODATE)) {
248         throw new DateTimeException("JapaneseDate before Meiji 6 are not supported");
249     }

```

```

250     for (int i = KNOWN_ERAS.length - 1; i > 0; i--) {
251         JapaneseEra era = KNOWN_ERAS[i];
252         if (date.compareTo(era.since) >= 0) {
253             return era;
254         }
255     }
256     return null;
257 }
258
259 static JapaneseEra toJapaneseEra(sun.util.calendar.Era privateEra) {
260     for (int i = ERA_CONFIG.length - 1; i >= 0; i--) {
261         if (ERA_CONFIG[i].equals(privateEra)) {
262             return KNOWN_ERAS[i];
263         }
264     }
265     return null;
266 }
267
268 static sun.util.calendar.Era privateEraFrom(LocalDate isoDate) {
269     for (int i = KNOWN_ERAS.length - 1; i > 0; i--) {
270         JapaneseEra era = KNOWN_ERAS[i];
271         if (isoDate.compareTo(era.since) >= 0) {
272             return ERA_CONFIG[i];
273         }
274     }
275     return null;
276 }
277
278 /**
279  * Returns the index into the arrays from the Era value.
280  * the eraValue is a valid Era number, -1..2.
281  *
282  * @param eraValue the era value to convert to the index
283  * @return the index of the current Era
284  */
285 private static int ordinal(int eraValue) {
286     return eraValue + ERA_OFFSET - 1;
287 }
288
289 //-----
290 /**
291  * Gets the numeric era {@code int} value.
292  * <p>
293  * The {@link #SHOWA} era that contains 1970-01-01 (ISO calendar system) has the value 1.
294  * Later eras are numbered from 2 ({@link #HEISEI}).
295  * Earlier eras are numbered 0 ({@link #TAISHO}), -1 ({@link #MEIJI}).
296  *
297  * @return the era value
298  */
299 @Override
300 public int getValue() {
301     return eraValue;
302 }

```

```

303
304 //-----
305 /**
306  * Gets the range of valid values for the specified field.
307  * <p>
308  * The range object expresses the minimum and maximum valid values for a field.
309  * This era is used to enhance the accuracy of the returned range.
310  * If it is not possible to return the range, because the field is not supported
311  * or for some other reason, an exception is thrown.
312  * <p>
313  * If the field is a {@link ChronoField} then the query is implemented here.
314  * The {@code ERA} field returns the range.
315  * All other {@code ChronoField} instances will throw an {@code
      UnsupportedTemporalTypeException}.
316  * <p>
317  * If the field is not a {@code ChronoField}, then the result of this method
318  * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor)}
319  * passing {@code this} as the argument.
320  * Whether the range can be obtained is determined by the field.
321  * <p>
322  * The range of valid Japanese eras can change over time due to the nature
323  * of the Japanese calendar system.
324  *
325  * @param field the field to query the range for, not null
326  * @return the range of valid values for the field, not null
327  * @throws DateTimeException if the range for the field cannot be obtained
328  * @throws UnsupportedTemporalTypeException if the unit is not supported
329  */
330 @Override // override as super would return range from 0 to 1
331 public ValueRange range(TemporalField field) {
332     if (field == ERA) {
333         return JapaneseChronology.INSTANCE.range(ERA);
334     }
335     return Era.super.range(field);
336 }
337
338 //-----
339 String getAbbreviation() {
340     int index = ordinal(getValue());
341     if (index == 0) {
342         return "";
343     }
344     return ERA_CONFIG[index].getAbbreviation();
345 }
346
347 String getName() {
348     return ERA_CONFIG[ordinal(getValue())].getName();
349 }
350
351 @Override
352 public String toString() {
353     return getName();
354 }

```

```

355 //-----
356 /**
357  * Defend against malicious streams.
358  *
359  * @param s the stream to read
360  * @throws InvalidObjectException always
361  */
362 private void readObject(ObjectInputStream s) throws InvalidObjectException {
363     throw new InvalidObjectException("Deserialization via serialization delegate");
364 }
365
366 //-----
367 /**
368  * Writes the object using a
369  * <a href="../../../serialized-form.html#java.time.chrono.Ser">dedicated serialized form</a>.
370  * @serialData
371  * <pre>
372  *   out.writeByte(5);          // identifies a JapaneseEra
373  *   out.writeInt(getValue());
374  * </pre>
375  *
376  * @return the instance of {@code Ser}, not null
377  */
378 private Object writeReplace() {
379     return new Ser(Ser.JAPANESE_ERA_TYPE, this);
380 }
381
382 void writeExternal(DataOutput out) throws IOException {
383     out.writeByte(this.getValue());
384 }
385
386 static JapaneseEra readExternal(DataInput in) throws IOException {
387     byte eraValue = in.readByte();
388     return JapaneseEra.of(eraValue);
389 }
390 }
391
392 }

```

5.20 MinguoChronology.java

Secuencia 90: java/time/chrono/MinguoChronology.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *

```

```
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
28  *
29  * All rights reserved.
30  *
31  * Redistribution and use in source and binary forms, with or without
32  * modification, are permitted provided that the following conditions are met:
33  *
34  * * Redistributions of source code must retain the above copyright notice,
35  *   this list of conditions and the following disclaimer.
36  *
37  * * Redistributions in binary form must reproduce the above copyright notice,
38  *   this list of conditions and the following disclaimer in the documentation
39  *   and/or other materials provided with the distribution.
40  *
41  * * Neither the name of JSR-310 nor the names of its contributors
42  *   may be used to endorse or promote products derived from this software
43  *   without specific prior written permission.
44  *
45  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56  */
57  package java.time.chrono;
58
59  import java.io.InvalidObjectException;
60  import static java.time.temporal.ChronoField.PROLEPTIC_MONTH;
61  import static java.time.temporal.ChronoField.YEAR;
62
63  import java.io.ObjectInputStream;
64  import java.io.Serializable;
```

```

65 import java.time.Clock;
66 import java.time.DateTimeException;
67 import java.time.Instant;
68 import java.time.LocalDate;
69 import java.time.ZoneId;
70 import java.time.format.ResolverStyle;
71 import java.time.temporal.ChronoField;
72 import java.time.temporal.TemporalAccessor;
73 import java.time.temporal.TemporalField;
74 import java.time.temporal.ValueRange;
75 import java.util.Arrays;
76 import java.util.List;
77 import java.util.Locale;
78 import java.util.Map;
79
80 /**
81  * The Minguo calendar system.
82  * <p>
83  * This chronology defines the rules of the Minguo calendar system.
84  * This calendar system is primarily used in the Republic of China, often known as Taiwan.
85  * Dates are aligned such that {@code 0001-01-01 (Minguo)} is {@code 1912-01-01 (ISO)}.
86  * <p>
87  * The fields are defined as follows:
88  * <ul>
89  * <li>era - There are two eras, the current 'Republic' (ERA_ROC) and the previous era (
      ERA_BEFORE_ROC).
90  * <li>year-of-era - The year-of-era for the current era increases uniformly from the epoch at year
      one.
91  * For the previous era the year increases from one as time goes backwards.
92  * The value for the current era is equal to the ISO proleptic-year minus 1911.
93  * <li>proleptic-year - The proleptic year is the same as the year-of-era for the
94  * current era. For the previous era, years have zero, then negative values.
95  * The value is equal to the ISO proleptic-year minus 1911.
96  * <li>month-of-year - The Minguo month-of-year exactly matches ISO.
97  * <li>day-of-month - The Minguo day-of-month exactly matches ISO.
98  * <li>day-of-year - The Minguo day-of-year exactly matches ISO.
99  * <li>leap-year - The Minguo leap-year pattern exactly matches ISO, such that the two calendars
100  * are never out of step.
101  * </ul>
102  *
103  * @implSpec
104  * This class is immutable and thread-safe.
105  *
106  * @since 1.8
107  */
108 public final class MinguoChronology extends AbstractChronology implements Serializable {
109
110     /**
111      * Singleton instance for the Minguo chronology.
112      */
113     public static final MinguoChronology INSTANCE = new MinguoChronology();
114
115     /**

```

```

116     * Serialization version.
117     */
118     private static final long serialVersionUID = 1039765215346859963L;
119     /**
120     * The difference in years between ISO and Minguo.
121     */
122     static final int YEARS_DIFFERENCE = 1911;
123
124     /**
125     * Restricted constructor.
126     */
127     private MinguoChronology() {
128     }
129
130     //-----
131     /**
132     * Gets the ID of the chronology - 'Minguo'.
133     * <p>
134     * The ID uniquely identifies the {@code Chronology}.
135     * It can be used to lookup the {@code Chronology} using {@link Chronology#of(String)}.
136     *
137     * @return the chronology ID - 'Minguo'
138     * @see #getCalendarType()
139     */
140     @Override
141     public String getId() {
142         return "Minguo";
143     }
144
145     /**
146     * Gets the calendar type of the underlying calendar system - 'roc'.
147     * <p>
148     * The calendar type is an identifier defined by the
149     * <em>Unicode Locale Data Markup Language (LDML)</em> specification.
150     * It can be used to lookup the {@code Chronology} using {@link Chronology#of(String)}.
151     * It can also be used as part of a locale, accessible via
152     * {@link Locale#getUnicodeLocaleType(String)} with the key 'ca'.
153     *
154     * @return the calendar system type - 'roc'
155     * @see #getId()
156     */
157     @Override
158     public String getCalendarType() {
159         return "roc";
160     }
161
162     //-----
163     /**
164     * Obtains a local date in Minguo calendar system from the
165     * era, year-of-era, month-of-year and day-of-month fields.
166     *
167     * @param era the Minguo era, not null
168     * @param yearOfEra the year-of-era

```



```

169     * @param month the month-of-year
170     * @param dayOfMonth the day-of-month
171     * @return the Minguo local date, not null
172     * @throws DateTimeException if unable to create the date
173     * @throws ClassCastException if the {@code era} is not a {@code MinguoEra}
174     */
175     @Override
176     public MinguoDate date(Era era, int yearOfEra, int month, int dayOfMonth) {
177         return date(prolepticYear(era, yearOfEra), month, dayOfMonth);
178     }
179
180     /**
181     * Obtains a local date in Minguo calendar system from the
182     * proleptic-year, month-of-year and day-of-month fields.
183     *
184     * @param prolepticYear the proleptic-year
185     * @param month the month-of-year
186     * @param dayOfMonth the day-of-month
187     * @return the Minguo local date, not null
188     * @throws DateTimeException if unable to create the date
189     */
190     @Override
191     public MinguoDate date(int prolepticYear, int month, int dayOfMonth) {
192         return new MinguoDate(LocalDate.of(prolepticYear + YEARS_DIFFERENCE, month, dayOfMonth));
193     }
194
195     /**
196     * Obtains a local date in Minguo calendar system from the
197     * era, year-of-era and day-of-year fields.
198     *
199     * @param era the Minguo era, not null
200     * @param yearOfEra the year-of-era
201     * @param dayOfYear the day-of-year
202     * @return the Minguo local date, not null
203     * @throws DateTimeException if unable to create the date
204     * @throws ClassCastException if the {@code era} is not a {@code MinguoEra}
205     */
206     @Override
207     public MinguoDate dateYearDay(Era era, int yearOfEra, int dayOfYear) {
208         return dateYearDay(prolepticYear(era, yearOfEra), dayOfYear);
209     }
210
211     /**
212     * Obtains a local date in Minguo calendar system from the
213     * proleptic-year and day-of-year fields.
214     *
215     * @param prolepticYear the proleptic-year
216     * @param dayOfYear the day-of-year
217     * @return the Minguo local date, not null
218     * @throws DateTimeException if unable to create the date
219     */
220     @Override
221     public MinguoDate dateYearDay(int prolepticYear, int dayOfYear) {

```

```
222     return new MinguoDate(LocalDate.ofYearDay(prolepticYear + YEARS_DIFFERENCE, dayOfYear));
223 }
224
225 /**
226  * Obtains a local date in the Minguo calendar system from the epoch-day.
227  *
228  * @param epochDay the epoch day
229  * @return the Minguo local date, not null
230  * @throws DateTimeException if unable to create the date
231  */
232 @Override // override with covariant return type
233 public MinguoDate dateEpochDay(long epochDay) {
234     return new MinguoDate(LocalDate.ofEpochDay(epochDay));
235 }
236
237 @Override
238 public MinguoDate dateNow() {
239     return dateNow(Clock.systemDefaultZone());
240 }
241
242 @Override
243 public MinguoDate dateNow(ZoneId zone) {
244     return dateNow(Clock.system(zone));
245 }
246
247 @Override
248 public MinguoDate dateNow(Clock clock) {
249     return date(LocalDate.now(clock));
250 }
251
252 @Override
253 public MinguoDate date(TemporalAccessor temporal) {
254     if (temporal instanceof MinguoDate) {
255         return (MinguoDate) temporal;
256     }
257     return new MinguoDate(LocalDate.from(temporal));
258 }
259
260 @Override
261 @SuppressWarnings("unchecked")
262 public ChronoLocalDateTime<MinguoDate> localDateTime(TemporalAccessor temporal) {
263     return (ChronoLocalDateTime<MinguoDate>)super.localDateTime(temporal);
264 }
265
266 @Override
267 @SuppressWarnings("unchecked")
268 public ChronoZonedDateTime<MinguoDate> zonedDateTime(TemporalAccessor temporal) {
269     return (ChronoZonedDateTime<MinguoDate>)super.zonedDateTime(temporal);
270 }
271
272 @Override
273 @SuppressWarnings("unchecked")
274 public ChronoZonedDateTime<MinguoDate> zonedDateTime(Instant instant, ZoneId zone) {
```

```

275         return (ChronoZonedDateTime<MinguoDate>)super.zonedDateTime(instant, zone);
276     }
277
278     //-----
279     /**
280      * Checks if the specified year is a leap year.
281      * <p>
282      * Minguo leap years occur exactly in line with ISO leap years.
283      * This method does not validate the year passed in, and only has a
284      * well-defined result for years in the supported range.
285      *
286      * @param prolepticYear the proleptic-year to check, not validated for range
287      * @return true if the year is a leap year
288      */
289     @Override
290     public boolean isLeapYear(long prolepticYear) {
291         return IsoChronology.INSTANCE.isLeapYear(prolepticYear + YEARS_DIFFERENCE);
292     }
293
294     @Override
295     public int prolepticYear(Era era, int yearOfEra) {
296         if (era instanceof MinguoEra == false) {
297             throw new ClassCastException("Era must be MinguoEra");
298         }
299         return (era == MinguoEra.ROC ? yearOfEra : 1 - yearOfEra);
300     }
301
302     @Override
303     public MinguoEra eraOf(int eraValue) {
304         return MinguoEra.of(eraValue);
305     }
306
307     @Override
308     public List<Era> eras() {
309         return Arrays.<Era>asList(MinguoEra.values());
310     }
311
312     //-----
313     @Override
314     public ValueRange range(ChronoField field) {
315         switch (field) {
316             case PROLEPTIC_MONTH: {
317                 ValueRange range = PROLEPTIC_MONTH.range();
318                 return ValueRange.of(range.getMinimum() - YEARS_DIFFERENCE * 12L, range.getMaximum()
319                     - YEARS_DIFFERENCE * 12L);
320             }
321             case YEAR_OF_ERA: {
322                 ValueRange range = YEAR.range();
323                 return ValueRange.of(1, range.getMaximum() - YEARS_DIFFERENCE, -range.getMinimum()
324                     + 1 + YEARS_DIFFERENCE);
325             }
326             case YEAR: {
327                 ValueRange range = YEAR.range();

```

```

326         return ValueRange.of(range.getMinimum() - YEARS_DIFFERENCE, range.getMaximum() -
327             YEARS_DIFFERENCE);
328     }
329     }
330     return field.range();
331 }
332
333 //-----
334 @Override // override for return type
335 public MinguoDate resolveDate(Map<TemporalField, Long> fieldValues, ResolverStyle resolverStyle
336     ) {
337     return (MinguoDate) super.resolveDate(fieldValues, resolverStyle);
338 }
339
340 //-----
341 /**
342  * Writes the Chronology using a
343  * <a href="../../serialized-form.html#java.time.chrono.Ser">dedicated serialized form</a>.
344  * @serialData
345  * <pre>
346  *   out.writeByte(1);    // identifies a Chronology
347  *   out.writeUTF(getId());
348  * </pre>
349  *
350  * @return the instance of {@code Ser}, not null
351  */
352 @Override
353 Object writeReplace() {
354     return super.writeReplace();
355 }
356
357 /**
358  * Defend against malicious streams.
359  *
360  * @param s the stream to read
361  * @throws InvalidObjectException always
362  */
363 private void readObject(ObjectInputStream s) throws InvalidObjectException {
364     throw new InvalidObjectException("Deserialization via serialization delegate");
365 }
366 }

```

5.21 MinguoDate.java

Secuencia 91: java/time/chrono/MinguoDate.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *

```

```
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27  * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
28  *
29  * All rights reserved.
30  *
31  * Redistribution and use in source and binary forms, with or without
32  * modification, are permitted provided that the following conditions are met:
33  *
34  * * Redistributions of source code must retain the above copyright notice,
35  *   this list of conditions and the following disclaimer.
36  *
37  * * Redistributions in binary form must reproduce the above copyright notice,
38  *   this list of conditions and the following disclaimer in the documentation
39  *   and/or other materials provided with the distribution.
40  *
41  * * Neither the name of JSR-310 nor the names of its contributors
42  *   may be used to endorse or promote products derived from this software
43  *   without specific prior written permission.
44  *
45  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56  */
57 package java.time.chrono;
58
59 import static java.time.chrono.MinguoChronology.YEARS_DIFFERENCE;
60 import static java.time.temporal.ChronoField.DAY_OF_MONTH;
61 import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
```

```

62 import static java.time.temporal.ChronoField.YEAR;
63
64 import java.io.DataInput;
65 import java.io.DataOutput;
66 import java.io.IOException;
67 import java.io.InvalidObjectException;
68 import java.io.ObjectInputStream;
69 import java.io.Serializable;
70 import java.time.Clock;
71 import java.time.DateTimeException;
72 import java.time.LocalDate;
73 import java.time.LocalDateTime;
74 import java.time.Period;
75 import java.time.ZoneId;
76 import java.time.temporal.ChronoField;
77 import java.time.temporal.TemporalAccessor;
78 import java.time.temporal.TemporalAdjuster;
79 import java.time.temporal.TemporalAmount;
80 import java.time.temporal.TemporalField;
81 import java.time.temporal.TemporalQuery;
82 import java.time.temporal.TemporalUnit;
83 import java.time.temporal.UnsupportedTemporalTypeException;
84 import java.time.temporal.ValueRange;
85 import java.util.Objects;
86
87 /**
88  * A date in the Minguo calendar system.
89  * <p>
90  * This date operates using the {@linkplain MinguoChronology Minguo calendar}.
91  * This calendar system is primarily used in the Republic of China, often known as Taiwan.
92  * Dates are aligned such that {@code 0001-01-01 (Minguo)} is {@code 1912-01-01 (ISO)}.
93  *
94  * <p>
95  * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
96  * class; use of identity-sensitive operations (including reference equality
97  * ({@code ==}), identity hash code, or synchronization) on instances of
98  * {@code MinguoDate} may have unpredictable results and should be avoided.
99  * The {@code equals} method should be used for comparisons.
100  *
101  * @implSpec
102  * This class is immutable and thread-safe.
103  *
104  * @since 1.8
105  */
106 public final class MinguoDate
107     extends ChronoLocalDateImpl<MinguoDate>
108     implements ChronoLocalDate, Serializable {
109
110     /**
111      * Serialization version.
112      */
113     private static final long serialVersionUID = 1300372329181994526L;
114

```

```

115  /**
116   * The underlying date.
117   */
118  private final transient LocalDate isoDate;
119
120  //-----
121  /**
122   * Obtains the current {@code MinguoDate} from the system clock in the default time-zone.
123   * <p>
124   * This will query the {@link Clock#systemDefaultZone() system clock} in the default
125   * time-zone to obtain the current date.
126   * <p>
127   * Using this method will prevent the ability to use an alternate clock for testing
128   * because the clock is hard-coded.
129   *
130   * @return the current date using the system clock and default time-zone, not null
131   */
132  public static MinguoDate now() {
133      return now(Clock.systemDefaultZone());
134  }
135
136  /**
137   * Obtains the current {@code MinguoDate} from the system clock in the specified time-zone.
138   * <p>
139   * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current date.
140   * Specifying the time-zone avoids dependence on the default time-zone.
141   * <p>
142   * Using this method will prevent the ability to use an alternate clock for testing
143   * because the clock is hard-coded.
144   *
145   * @param zone the zone ID to use, not null
146   * @return the current date using the system clock, not null
147   */
148  public static MinguoDate now(ZoneId zone) {
149      return now(Clock.system(zone));
150  }
151
152  /**
153   * Obtains the current {@code MinguoDate} from the specified clock.
154   * <p>
155   * This will query the specified clock to obtain the current date - today.
156   * Using this method allows the use of an alternate clock for testing.
157   * The alternate clock may be introduced using {@linkplain Clock dependency injection}.
158   *
159   * @param clock the clock to use, not null
160   * @return the current date, not null
161   * @throws DateTimeException if the current date cannot be obtained
162   */
163  public static MinguoDate now(Clock clock) {
164      return new MinguoDate(LocalDate.now(clock));
165  }
166
167  /**

```

```

168 * Obtains a {@code MinguoDate} representing a date in the Minguo calendar
169 * system from the proleptic-year, month-of-year and day-of-month fields.
170 * <p>
171 * This returns a {@code MinguoDate} with the specified fields.
172 * The day must be valid for the year and month, otherwise an exception will be thrown.
173 *
174 * @param prolepticYear the Minguo proleptic-year
175 * @param month the Minguo month-of-year, from 1 to 12
176 * @param dayOfMonth the Minguo day-of-month, from 1 to 31
177 * @return the date in Minguo calendar system, not null
178 * @throws DateTimeException if the value of any field is out of range,
179 * or if the day-of-month is invalid for the month-year
180 */
181 public static MinguoDate of(int prolepticYear, int month, int dayOfMonth) {
182     return new MinguoDate(LocalDate.of(prolepticYear + YEARS_DIFFERENCE, month, dayOfMonth));
183 }
184
185 /**
186 * Obtains a {@code MinguoDate} from a temporal object.
187 * <p>
188 * This obtains a date in the Minguo calendar system based on the specified temporal.
189 * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
190 * which this factory converts to an instance of {@code MinguoDate}.
191 * <p>
192 * The conversion typically uses the {@link ChronoField#EPOCH_DAY EPOCH_DAY}
193 * field, which is standardized across calendar systems.
194 * <p>
195 * This method matches the signature of the functional interface {@link TemporalQuery}
196 * allowing it to be used as a query via method reference, {@code MinguoDate::from}.
197 *
198 * @param temporal the temporal object to convert, not null
199 * @return the date in Minguo calendar system, not null
200 * @throws DateTimeException if unable to convert to a {@code MinguoDate}
201 */
202 public static MinguoDate from(TemporalAccessor temporal) {
203     return MinguoChronology.INSTANCE.date(temporal);
204 }
205
206 //-----
207 /**
208 * Creates an instance from an ISO date.
209 *
210 * @param isoDate the standard local date, validated not null
211 */
212 MinguoDate(LocalDate isoDate) {
213     Objects.requireNonNull(isoDate, "isoDate");
214     this.isoDate = isoDate;
215 }
216
217 //-----
218 /**
219 * Gets the chronology of this date, which is the Minguo calendar system.
220 * <p>

```



```

273         }
274         return getChronology().range(f);
275     }
276     throw new UnsupportedOperationException("Unsupported field: " + field);
277 }
278 return field.rangeRefinedBy(this);
279 }
280
281 @Override
282 public long getLong(TemporalField field) {
283     if (field instanceof ChronoField) {
284         switch ((ChronoField) field) {
285             case PROLEPTIC_MONTH:
286                 return getProlepticMonth();
287             case YEAR_OF_ERA: {
288                 int prolepticYear = getProlepticYear();
289                 return (prolepticYear >= 1 ? prolepticYear : 1 - prolepticYear);
290             }
291             case YEAR:
292                 return getProlepticYear();
293             case ERA:
294                 return (getProlepticYear() >= 1 ? 1 : 0);
295         }
296         return isoDate.getLong(field);
297     }
298     return field.getFrom(this);
299 }
300
301 private long getProlepticMonth() {
302     return getProlepticYear() * 12L + isoDate.getMonthValue() - 1;
303 }
304
305 private int getProlepticYear() {
306     return isoDate.getYear() - YEARS_DIFFERENCE;
307 }
308
309 //-----
310 @Override
311 public MinguoDate with(TemporalField field, long newValue) {
312     if (field instanceof ChronoField) {
313         ChronoField f = (ChronoField) field;
314         if (getLong(f) == newValue) {
315             return this;
316         }
317         switch (f) {
318             case PROLEPTIC_MONTH:
319                 getChronology().range(f).checkValidValue(newValue, f);
320                 return plusMonths(newValue - getProlepticMonth());
321             case YEAR_OF_ERA:
322             case YEAR:
323             case ERA: {
324                 int nvalue = getChronology().range(f).checkValidIntValue(newValue, f);
325                 switch (f) {

```

```

326         case YEAR_OF_ERA:
327             return with(isoDate.withYear(getProlepticYear() >= 1 ? nvalue +
328                                     YEARS_DIFFERENCE : (1 - nvalue) + YEARS_DIFFERENCE));
329         case YEAR:
330             return with(isoDate.withYear(nvalue + YEARS_DIFFERENCE));
331         case ERA:
332             return with(isoDate.withYear((1 - getProlepticYear()) +
333                                     YEARS_DIFFERENCE));
334     }
335     }
336     return with(isoDate.with(field, newValue));
337 }
338 return super.with(field, newValue);
339 }
340 /**
341  * {@inheritDoc}
342  * @throws DateTimeException {@inheritDoc}
343  * @throws ArithmeticException {@inheritDoc}
344  */
345 @Override
346 public MinguoDate with(TemporalAdjuster adjuster) {
347     return super.with(adjuster);
348 }
349
350 /**
351  * {@inheritDoc}
352  * @throws DateTimeException {@inheritDoc}
353  * @throws ArithmeticException {@inheritDoc}
354  */
355 @Override
356 public MinguoDate plus(TemporalAmount amount) {
357     return super.plus(amount);
358 }
359
360 /**
361  * {@inheritDoc}
362  * @throws DateTimeException {@inheritDoc}
363  * @throws ArithmeticException {@inheritDoc}
364  */
365 @Override
366 public MinguoDate minus(TemporalAmount amount) {
367     return super.minus(amount);
368 }
369
370 //-----
371 @Override
372 public MinguoDate plusYears(long years) {
373     return with(isoDate.plusYears(years));
374 }
375
376 @Override

```

```
377     MinguoDate plusMonths(long months) {
378         return with(isoDate.plusMonths(months));
379     }
380
381     @Override
382     MinguoDate plusWeeks(long weeksToAdd) {
383         return super.plusWeeks(weeksToAdd);
384     }
385
386     @Override
387     MinguoDate plusDays(long days) {
388         return with(isoDate.plusDays(days));
389     }
390
391     @Override
392     public MinguoDate plus(long amountToAdd, TemporalUnit unit) {
393         return super.plus(amountToAdd, unit);
394     }
395
396     @Override
397     public MinguoDate minus(long amountToAdd, TemporalUnit unit) {
398         return super.minus(amountToAdd, unit);
399     }
400
401     @Override
402     MinguoDate minusYears(long yearsToSubtract) {
403         return super.minusYears(yearsToSubtract);
404     }
405
406     @Override
407     MinguoDate minusMonths(long monthsToSubtract) {
408         return super.minusMonths(monthsToSubtract);
409     }
410
411     @Override
412     MinguoDate minusWeeks(long weeksToSubtract) {
413         return super.minusWeeks(weeksToSubtract);
414     }
415
416     @Override
417     MinguoDate minusDays(long daysToSubtract) {
418         return super.minusDays(daysToSubtract);
419     }
420
421     private MinguoDate with(LocalDate newDate) {
422         return (newDate.equals(isoDate) ? this : new MinguoDate(newDate));
423     }
424
425     @Override          // for javadoc and covariant return type
426     @SuppressWarnings("unchecked")
427     public final ChronoLocalDateTime<MinguoDate> atTime(LocalTime localTime) {
428         return (ChronoLocalDateTime<MinguoDate>)super.atTime(localTime);
429     }
```

```

430
431 @Override
432 public ChronoPeriod until(ChronoLocalDate endDate) {
433     Period period = isoDate.until(endDate);
434     return getChronology().period(period.getYears(), period.getMonths(), period.getDays());
435 }
436
437 @Override // override for performance
438 public long toEpochDay() {
439     return isoDate.toEpochDay();
440 }
441
442 //-----
443 /**
444  * Compares this date to another date, including the chronology.
445  * <p>
446  * Compares this {@code MinguoDate} with another ensuring that the date is the same.
447  * <p>
448  * Only objects of type {@code MinguoDate} are compared, other types return false.
449  * To compare the dates of two {@code TemporalAccessor} instances, including dates
450  * in two different chronologies, use {@link ChronoField#EPOCH_DAY} as a comparator.
451  *
452  * @param obj the object to check, null returns false
453  * @return true if this is equal to the other date
454  */
455 @Override // override for performance
456 public boolean equals(Object obj) {
457     if (this == obj) {
458         return true;
459     }
460     if (obj instanceof MinguoDate) {
461         MinguoDate otherDate = (MinguoDate) obj;
462         return this.isoDate.equals(otherDate.isoDate);
463     }
464     return false;
465 }
466
467 /**
468  * A hash code for this date.
469  *
470  * @return a suitable hash code based only on the Chronology and the date
471  */
472 @Override // override for performance
473 public int hashCode() {
474     return getChronology().getId().hashCode() ^ isoDate.hashCode();
475 }
476
477 //-----
478 /**
479  * Defend against malicious streams.
480  *
481  * @param s the stream to read
482  * @throws InvalidObjectException always

```

```

483     */
484     private void readObject(ObjectInputStream s) throws InvalidObjectException {
485         throw new InvalidObjectException("Deserialization via serialization delegate");
486     }
487
488     /**
489     * Writes the object using a
490     * <a href="../../serialized-form.html#java.time.chrono.Ser">dedicated serialized form</a>.
491     * @serialData
492     * <pre>
493     *   out.writeByte(8);           // identifies a MinguoDate
494     *   out.writeInt(get(YEAR));
495     *   out.writeByte(get(MONTH_OF_YEAR));
496     *   out.writeByte(get(DAY_OF_MONTH));
497     * </pre>
498     *
499     * @return the instance of {@code Ser}, not null
500     */
501     private Object writeReplace() {
502         return new Ser(Ser.MINGUO_DATE_TYPE, this);
503     }
504
505     void writeExternal(DataOutput out) throws IOException {
506         // MinguoChronology is implicit in the MINGUO_DATE_TYPE
507         out.writeInt(get(YEAR));
508         out.writeByte(get(MONTH_OF_YEAR));
509         out.writeByte(get(DAY_OF_MONTH));
510     }
511
512     static MinguoDate readExternal(DataInput in) throws IOException {
513         int year = in.readInt();
514         int month = in.readByte();
515         int dayOfMonth = in.readByte();
516         return MinguoChronology.INSTANCE.date(year, month, dayOfMonth);
517     }
518
519 }

```

5.22 MinguoEra.java

Secuencia 92: java/time/chrono/MinguoEra.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *

```

```
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26 /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62 package java.time.chrono;
63
64 import java.time.DateTimeException;
65
```

```

66  /**
67   * An era in the Minguo calendar system.
68   * <p>
69   * The Minguo calendar system has two eras.
70   * The current era, for years from 1 onwards, is known as the 'Republic of China' era.
71   * All previous years, zero or earlier in the proleptic count or one and greater
72   * in the year-of-era count, are part of the 'Before Republic of China' era.
73   *
74   * <table summary="Minguo years and eras" cellpadding="2" cellspacing="3" border="0" >
75   * <thead>
76   * <tr class="tableSubHeadingColor">
77   * <th class="colFirst" align="left">year-of-era</th>
78   * <th class="colFirst" align="left">era</th>
79   * <th class="colFirst" align="left">proleptic-year</th>
80   * <th class="colLast" align="left">ISO proleptic-year</th>
81   * </tr>
82   * </thead>
83   * <tbody>
84   * <tr class="rowColor">
85   * <td>2</td><td>ROC</td><td>2</td><td>1913</td>
86   * </tr>
87   * <tr class="altColor">
88   * <td>1</td><td>ROC</td><td>1</td><td>1912</td>
89   * </tr>
90   * <tr class="rowColor">
91   * <td>1</td><td>BEFORE_ROC</td><td>0</td><td>1911</td>
92   * </tr>
93   * <tr class="altColor">
94   * <td>2</td><td>BEFORE_ROC</td><td>-1</td><td>1910</td>
95   * </tr>
96   * </tbody>
97   * </table>
98   * <p>
99   * <b>Do not use {@code ordinal()} to obtain the numeric representation of {@code MinguoEra}.
100  * Use {@code getValue()} instead.</b>
101  *
102  * @implSpec
103  * This is an immutable and thread-safe enum.
104  *
105  * @since 1.8
106  */
107 public enum MinguoEra implements Era {
108
109     /**
110      * The singleton instance for the era before the current one, 'Before Republic of China Era',
111      * which has the numeric value 0.
112      */
113     BEFORE_ROC,
114
115     /**
116      * The singleton instance for the current era, 'Republic of China Era',
117      * which has the numeric value 1.
118      */
119     ROC;

```



```

119
120 //-----
121 /**
122  * Obtains an instance of {@code MinguoEra} from an {@code int} value.
123  * <p>
124  * {@code MinguoEra} is an enum representing the Minguo eras of BEFORE_ROC/ROC.
125  * This factory allows the enum to be obtained from the {@code int} value.
126  *
127  * @param minguoEra the BEFORE_ROC/ROC value to represent, from 0 (BEFORE_ROC) to 1 (ROC)
128  * @return the era singleton, not null
129  * @throws DateTimeException if the value is invalid
130  */
131 public static MinguoEra of(int minguoEra) {
132     switch (minguoEra) {
133         case 0:
134             return BEFORE_ROC;
135         case 1:
136             return ROC;
137         default:
138             throw new DateTimeException("Invalid era: " + minguoEra);
139     }
140 }
141
142 //-----
143 /**
144  * Gets the numeric era {@code int} value.
145  * <p>
146  * The era BEFORE_ROC has the value 0, while the era ROC has the value 1.
147  *
148  * @return the era value, from 0 (BEFORE_ROC) to 1 (ROC)
149  */
150 @Override
151 public int getValue() {
152     return ordinal();
153 }
154
155 }

```

5.23 Ser.java

Secuencia 93: java/time/chrono/Ser.java

```

1 /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *

```

```
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26 /*
27  * Copyright (c) 2011-2012, Stephen Colebourne & Michael Nascimento Santos
28  *
29  * All rights reserved.
30  *
31  * Redistribution and use in source and binary forms, with or without
32  * modification, are permitted provided that the following conditions are met:
33  *
34  * * Redistributions of source code must retain the above copyright notice,
35  *   this list of conditions and the following disclaimer.
36  *
37  * * Redistributions in binary form must reproduce the above copyright notice,
38  *   this list of conditions and the following disclaimer in the documentation
39  *   and/or other materials provided with the distribution.
40  *
41  * * Neither the name of JSR-310 nor the names of its contributors
42  *   may be used to endorse or promote products derived from this software
43  *   without specific prior written permission.
44  *
45  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56  */
57 package java.time.chrono;
58
59 import java.io.Externalizable;
60 import java.io.IOException;
61 import java.io.InvalidClassException;
62 import java.io.ObjectInput;
63 import java.io.ObjectOutput;
64 import java.io.StreamCorruptedException;
65 import java.time.LocalDate;
```

```

66 import java.time.LocalDateTime;
67
68 /**
69  * The shared serialization delegate for this package.
70  *
71  * @implNote
72  * This class wraps the object being serialized, and takes a byte representing the type of the
73  * class to
74  * be serialized. This byte can also be used for versioning the serialization format. In this
75  * case another
76  * byte flag would be used in order to specify an alternative version of the type format.
77  * For example {@code CHRONO_TYPE_VERSION_2 = 21}
78  * <p>
79  * In order to serialize the object it writes its byte and then calls back to the appropriate class
80  * where
81  * the serialization is performed. In order to deserialize the object it read in the type byte,
82  * switching
83  * in order to select which class to call back into.
84  * <p>
85  * The serialization format is determined on a per class basis. In the case of field based classes
86  * each
87  * of the fields is written out with an appropriate size format in descending order of the field's
88  * size. For
89  * example in the case of {@link LocalDate} year is written before month. Composite classes, such
90  * as
91  * {@link LocalDateTime} are serialized as one object. Enum classes are serialized using the index
92  * of their
93  * element.
94  * <p>
95  * This class is mutable and should be created once per serialization.
96  *
97  * @serial include
98  * @since 1.8
99  */
100 final class Ser implements Externalizable {
101
102     /**
103      * Serialization version.
104      */
105     private static final long serialVersionUID = -6103370247208168577L;
106
107     static final byte CHRONO_TYPE = 1;
108     static final byte CHRONO_LOCAL_DATE_TIME_TYPE = 2;
109     static final byte CHRONO_ZONE_DATE_TIME_TYPE = 3;
110     static final byte JAPANESE_DATE_TYPE = 4;
111     static final byte JAPANESE_ERA_TYPE = 5;
112     static final byte HIJRAH_DATE_TYPE = 6;
113     static final byte MINGUO_DATE_TYPE = 7;
114     static final byte THAIBUDDHIST_DATE_TYPE = 8;
115     static final byte CHRONO_PERIOD_TYPE = 9;
116
117     /** The type being serialized. */
118     private byte type;

```

```

111  /** The object being serialized. */
112  private Object object;
113
114  /**
115   * Constructor for deserialization.
116   */
117  public Ser() {
118  }
119
120  /**
121   * Creates an instance for serialization.
122   *
123   * @param type the type
124   * @param object the object
125   */
126  Ser(byte type, Object object) {
127      this.type = type;
128      this.object = object;
129  }
130
131  //-----
132  /**
133   * Implements the {@code Externalizable} interface to write the object.
134   * @serialData
135   * Each serializable class is mapped to a type that is the first byte
136   * in the stream. Refer to each class {@code writeReplace}
137   * serialized form for the value of the type and sequence of values for the type.
138   * <ul>
139   * <li><a href="../../serialized-form.html#java.time.chrono.HijrahChronology">
140   *     HijrahChronology.writeReplace</a>
141   * <li><a href="../../serialized-form.html#java.time.chrono.IsoChronology">IsoChronology.
142   *     writeReplace</a>
143   * <li><a href="../../serialized-form.html#java.time.chrono.JapaneseChronology">
144   *     JapaneseChronology.writeReplace</a>
145   * <li><a href="../../serialized-form.html#java.time.chrono.MinguoChronology">
146   *     MinguoChronology.writeReplace</a>
147   * <li><a href="../../serialized-form.html#java.time.chrono.ThaiBuddhistChronology">
148   *     ThaiBuddhistChronology.writeReplace</a>
149   * <li><a href="../../serialized-form.html#java.time.chrono.ChronoLocalDateTimeImpl">
150   *     ChronoLocalDateTime.writeReplace</a>
151   * <li><a href="../../serialized-form.html#java.time.chrono.ChronoZonedDateTimeImpl">
152   *     ChronoZonedDateTime.writeReplace</a>
153   * <li><a href="../../serialized-form.html#java.time.chrono.JapaneseDate">JapaneseDate.
154   *     writeReplace</a>
155   * <li><a href="../../serialized-form.html#java.time.chrono.JapaneseEra">JapaneseEra.
156   *     writeReplace</a>
157   * <li><a href="../../serialized-form.html#java.time.chrono.HijrahDate">HijrahDate.
158   *     writeReplace</a>
159   * <li><a href="../../serialized-form.html#java.time.chrono.MinguoDate">MinguoDate.
160   *     writeReplace</a>
161   * <li><a href="../../serialized-form.html#java.time.chrono.ThaiBuddhistDate">
162   *     ThaiBuddhistDate.writeReplace</a>
163   * </ul>

```

```

152     *
153     * @param out the data stream to write to, not null
154     */
155     @Override
156     public void writeExternal(ObjectOutput out) throws IOException {
157         writeInternal(type, object, out);
158     }
159
160     private static void writeInternal(byte type, Object object, ObjectOutput out) throws
161         IOException {
162         out.writeByte(type);
163         switch (type) {
164             case CHRONO_TYPE:
165                 ((AbstractChronology) object).writeExternal(out);
166                 break;
167             case CHRONO_LOCAL_DATE_TIME_TYPE:
168                 ((ChronoLocalDateTimeImpl<?>) object).writeExternal(out);
169                 break;
170             case CHRONO_ZONE_DATE_TIME_TYPE:
171                 ((ChronoZonedDateTimeImpl<?>) object).writeExternal(out);
172                 break;
173             case JAPANESE_DATE_TYPE:
174                 ((JapaneseDate) object).writeExternal(out);
175                 break;
176             case JAPANESE_ERA_TYPE:
177                 ((JapaneseEra) object).writeExternal(out);
178                 break;
179             case HIJRAH_DATE_TYPE:
180                 ((HijrahDate) object).writeExternal(out);
181                 break;
182             case MINGUO_DATE_TYPE:
183                 ((MinguoDate) object).writeExternal(out);
184                 break;
185             case THAIBUDDHIST_DATE_TYPE:
186                 ((ThaiBuddhistDate) object).writeExternal(out);
187                 break;
188             case CHRONO_PERIOD_TYPE:
189                 ((ChronoPeriodImpl) object).writeExternal(out);
190                 break;
191             default:
192                 throw new InvalidClassException("Unknown serialized type");
193         }
194     }
195
196     //-----
197     /**
198     * Implements the {@code Externalizable} interface to read the object.
199     * @serialData
200     * The streamed type and parameters defined by the type's {@code writeReplace}
201     * method are read and passed to the corresponding static factory for the type
202     * to create a new instance. That instance is returned as the de-serialized
203     * {@code Ser} object.
204     */

```

```

204 * <ul>
205 * <li><a href="../../../serialized-form.html#java.time.chrono.HijrahChronology">
    HijrahChronology</a> - Chronology.of(id)
206 * <li><a href="../../../serialized-form.html#java.time.chrono.IsoChronology">IsoChronology</a>
    - Chronology.of(id)
207 * <li><a href="../../../serialized-form.html#java.time.chrono.JapaneseChronology">
    JapaneseChronology</a> - Chronology.of(id)
208 * <li><a href="../../../serialized-form.html#java.time.chrono.MinguoChronology">
    MinguoChronology</a> - Chronology.of(id)
209 * <li><a href="../../../serialized-form.html#java.time.chrono.ThaiBuddhistChronology">
    ThaiBuddhistChronology</a> - Chronology.of(id)
210 * <li><a href="../../../serialized-form.html#java.time.chrono.ChronoLocalDateTimeImpl">
    ChronoLocalDateTime</a> - date.atTime(time)
211 * <li><a href="../../../serialized-form.html#java.time.chrono.ChronoZonedDateTimeImpl">
    ChronoZonedDateTime</a> - dateTime.atZone(offset).withZoneSameLocal(zone)
212 * <li><a href="../../../serialized-form.html#java.time.chrono.JapaneseDate">JapaneseDate</a> -
    JapaneseChronology.INSTANCE.date(year, month, dayOfMonth)
213 * <li><a href="../../../serialized-form.html#java.time.chrono.JapaneseEra">JapaneseEra</a> -
    JapaneseEra.of(eraValue)
214 * <li><a href="../../../serialized-form.html#java.time.chrono.HijrahDate">HijrahDate</a> -
    HijrahChronology chrono.date(year, month, dayOfMonth)
215 * <li><a href="../../../serialized-form.html#java.time.chrono.MinguoDate">MinguoDate</a> -
    MinguoChronology.INSTANCE.date(year, month, dayOfMonth)
216 * <li><a href="../../../serialized-form.html#java.time.chrono.ThaiBuddhistDate">
    ThaiBuddhistDate</a> - ThaiBuddhistChronology.INSTANCE.date(year, month, dayOfMonth)
217 * </ul>
218 *
219 * @param in the data stream to read from, not null
220 */
221 @Override
222 public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
223     type = in.readByte();
224     object = readInternal(type, in);
225 }
226
227 static Object read(ObjectInput in) throws IOException, ClassNotFoundException {
228     byte type = in.readByte();
229     return readInternal(type, in);
230 }
231
232 private static Object readInternal(byte type, ObjectInput in) throws IOException,
    ClassNotFoundException {
233     switch (type) {
234         case CHRONO_TYPE: return AbstractChronology.readExternal(in);
235         case CHRONO_LOCAL_DATE_TIME_TYPE: return ChronoLocalDateTimeImpl.readExternal(in);
236         case CHRONO_ZONE_DATE_TIME_TYPE: return ChronoZonedDateTimeImpl.readExternal(in);
237         case JAPANESE_DATE_TYPE: return JapaneseDate.readExternal(in);
238         case JAPANESE_ERA_TYPE: return JapaneseEra.readExternal(in);
239         case HIJRAH_DATE_TYPE: return HijrahDate.readExternal(in);
240         case MINGUO_DATE_TYPE: return MinguoDate.readExternal(in);
241         case THAIBUDDHIST_DATE_TYPE: return ThaiBuddhistDate.readExternal(in);
242         case CHRONO_PERIOD_TYPE: return ChronoPeriodImpl.readExternal(in);
243         default: throw new StreamCorruptedException("Unknown serialized type");
    }

```

```

244     }
245 }
246
247 /**
248  * Returns the object that will replace this one.
249  *
250  * @return the read object, should never be null
251  */
252 private Object readResolve() {
253     return object;
254 }
255
256 }

```

5.24 ThaiBuddhistChronology.java

Secuencia 94: java/time/chrono/ThaiBuddhistChronology.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
28 *
29 * All rights reserved.
30 *
31 * Redistribution and use in source and binary forms, with or without
32 * modification, are permitted provided that the following conditions are met:
33 *
34 * * Redistributions of source code must retain the above copyright notice,
35 *   this list of conditions and the following disclaimer.
36 *

```

```

37  * * Redistributions in binary form must reproduce the above copyright notice,
38  *   this list of conditions and the following disclaimer in the documentation
39  *   and/or other materials provided with the distribution.
40  *
41  * * Neither the name of JSR-310 nor the names of its contributors
42  *   may be used to endorse or promote products derived from this software
43  *   without specific prior written permission.
44  *
45  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56  */
57 package java.time.chrono;
58
59 import java.io.InvalidObjectException;
60 import static java.time.temporal.ChronoField.PROLEPTIC_MONTH;
61 import static java.time.temporal.ChronoField.YEAR;
62
63 import java.io.ObjectInputStream;
64 import java.io.Serializable;
65 import java.time.Clock;
66 import java.time.DateTimeException;
67 import java.time.Instant;
68 import java.time.LocalDate;
69 import java.time.ZoneId;
70 import java.time.format.ResolverStyle;
71 import java.time.temporal.ChronoField;
72 import java.time.temporal.TemporalAccessor;
73 import java.time.temporal.TemporalField;
74 import java.time.temporal.ValueRange;
75 import java.util.Arrays;
76 import java.util.HashMap;
77 import java.util.List;
78 import java.util.Locale;
79 import java.util.Map;
80
81 /**
82  * The Thai Buddhist calendar system.
83  * <p>
84  * This chronology defines the rules of the Thai Buddhist calendar system.
85  * This calendar system is primarily used in Thailand.
86  * Dates are aligned such that {@code 2484-01-01 (Buddhist)} is {@code 1941-01-01 (ISO)}.
87  * <p>
88  * The fields are defined as follows:
89  * <ul>

```



```

90  * <li>era - There are two eras, the current 'Buddhist' (ERA_BE) and the previous era (
      ERA_BEFORE_BE).
91  * <li>year-of-era - The year-of-era for the current era increases uniformly from the epoch at year
      one.
92  * For the previous era the year increases from one as time goes backwards.
93  * The value for the current era is equal to the ISO proleptic-year plus 543.
94  * <li>proleptic-year - The proleptic year is the same as the year-of-era for the
95  * current era. For the previous era, years have zero, then negative values.
96  * The value is equal to the ISO proleptic-year plus 543.
97  * <li>month-of-year - The ThaiBuddhist month-of-year exactly matches ISO.
98  * <li>day-of-month - The ThaiBuddhist day-of-month exactly matches ISO.
99  * <li>day-of-year - The ThaiBuddhist day-of-year exactly matches ISO.
100 * <li>leap-year - The ThaiBuddhist leap-year pattern exactly matches ISO, such that the two
      calendars
101 * are never out of step.
102 * </ul>
103 *
104 * @implSpec
105 * This class is immutable and thread-safe.
106 *
107 * @since 1.8
108 */
109 public final class ThaiBuddhistChronology extends AbstractChronology implements Serializable {
110
111     /**
112      * Singleton instance of the Buddhist chronology.
113      */
114     public static final ThaiBuddhistChronology INSTANCE = new ThaiBuddhistChronology();
115
116     /**
117      * Serialization version.
118      */
119     private static final long serialVersionUID = 2775954514031616474L;
120
121     /**
122      * Containing the offset to add to the ISO year.
123      */
124     static final int YEARS_DIFFERENCE = 543;
125
126     /**
127      * Narrow names for eras.
128      */
129     private static final HashMap<String, String[]> ERA_NARROW_NAMES = new HashMap<>();
130
131     /**
132      * Short names for eras.
133      */
134     private static final HashMap<String, String[]> ERA_SHORT_NAMES = new HashMap<>();
135
136     /**
137      * Full names for eras.
138      */
139     private static final HashMap<String, String[]> ERA_FULL_NAMES = new HashMap<>();
140
141     /**
142      * Fallback language for the era names.
143      */
144     private static final String FALLBACK_LANGUAGE = "en";

```

```

140 /**
141  * Language that has the era names.
142  */
143 private static final String TARGET_LANGUAGE = "th";
144 /**
145  * Name data.
146  */
147 static {
148     ERA_NARROW_NAMES.put(FALLBACK_LANGUAGE, new String[]{"BB", "BE"});
149     ERA_NARROW_NAMES.put(TARGET_LANGUAGE, new String[]{"BB", "BE"});
150     ERA_SHORT_NAMES.put(FALLBACK_LANGUAGE, new String[]{"B.B.", "B.E."});
151     ERA_SHORT_NAMES.put(TARGET_LANGUAGE,
152         new String[]{"\u0e1e.\u0e28.",
153             "\u0e1b\u0e35\u0e01\u0e48\u0e2d\u0e19\u0e04\u0e23\u0e34\u0e2a\u0e15\u0e4c\u0e01\u0e32\u0e25\u0e17\u0e35\u0e48"});
154     ERA_FULL_NAMES.put(FALLBACK_LANGUAGE, new String[]{"Before Buddhist", "Buddhist Era"});
155     ERA_FULL_NAMES.put(TARGET_LANGUAGE,
156         new String[]{"\u0e1e\u0e38\u0e17\u0e18\u0e28\u0e31\u0e01\u0e23\u0e32\u0e0a",
157             "\u0e1b\u0e35\u0e01\u0e48\u0e2d\u0e19\u0e04\u0e23\u0e34\u0e2a\u0e15\u0e4c\u0e01\u0e32\u0e25\u0e17\u0e35\u0e48"});
158 }
159
160 /**
161  * Restricted constructor.
162  */
163 private ThaiBuddhistChronology() {
164 }
165
166 //-----
167 /**
168  * Gets the ID of the chronology - 'ThaiBuddhist'.
169  * <p>
170  * The ID uniquely identifies the {@code Chronology}.
171  * It can be used to lookup the {@code Chronology} using {@link Chronology#of(String)}.
172  *
173  * @return the chronology ID - 'ThaiBuddhist'
174  * @see #getCalendarType()
175  */
176 @Override
177 public String getId() {
178     return "ThaiBuddhist";
179 }
180
181 /**
182  * Gets the calendar type of the underlying calendar system - 'buddhist'.
183  * <p>
184  * The calendar type is an identifier defined by the
185  * <em>Unicode Locale Data Markup Language (LDML)</em> specification.
186  * It can be used to lookup the {@code Chronology} using {@link Chronology#of(String)}.
187  * It can also be used as part of a locale, accessible via
188  * {@link Locale#getUnicodeLocaleType(String)} with the key 'ca'.
189  *
190  * @return the calendar system type - 'buddhist'

```

```

191     * @see #getId()
192     */
193     @Override
194     public String getCalendarType() {
195         return "buddhist";
196     }
197
198     //-----
199     /**
200      * Obtains a local date in Thai Buddhist calendar system from the
201      * era, year-of-era, month-of-year and day-of-month fields.
202      *
203      * @param era the Thai Buddhist era, not null
204      * @param yearOfEra the year-of-era
205      * @param month the month-of-year
206      * @param dayOfMonth the day-of-month
207      * @return the Thai Buddhist local date, not null
208      * @throws DateTimeException if unable to create the date
209      * @throws ClassCastException if the {@code era} is not a {@code ThaiBuddhistEra}
210      */
211     @Override
212     public ThaiBuddhistDate date(Era era, int yearOfEra, int month, int dayOfMonth) {
213         return date(prolepticYear(era, yearOfEra), month, dayOfMonth);
214     }
215
216     /**
217      * Obtains a local date in Thai Buddhist calendar system from the
218      * proleptic-year, month-of-year and day-of-month fields.
219      *
220      * @param prolepticYear the proleptic-year
221      * @param month the month-of-year
222      * @param dayOfMonth the day-of-month
223      * @return the Thai Buddhist local date, not null
224      * @throws DateTimeException if unable to create the date
225      */
226     @Override
227     public ThaiBuddhistDate date(int prolepticYear, int month, int dayOfMonth) {
228         return new ThaiBuddhistDate(LocalDate.of(prolepticYear - YEARS_DIFFERENCE, month,
229             dayOfMonth));
230
231     /**
232      * Obtains a local date in Thai Buddhist calendar system from the
233      * era, year-of-era and day-of-year fields.
234      *
235      * @param era the Thai Buddhist era, not null
236      * @param yearOfEra the year-of-era
237      * @param dayOfYear the day-of-year
238      * @return the Thai Buddhist local date, not null
239      * @throws DateTimeException if unable to create the date
240      * @throws ClassCastException if the {@code era} is not a {@code ThaiBuddhistEra}
241      */
242     @Override

```

```

243 public ThaiBuddhistDate dateYearDay(Era era, int yearOfEra, int dayOfYear) {
244     return dateYearDay(prolepticYear(era, yearOfEra), dayOfYear);
245 }
246
247 /**
248  * Obtains a local date in Thai Buddhist calendar system from the
249  * proleptic-year and day-of-year fields.
250  *
251  * @param prolepticYear the proleptic-year
252  * @param dayOfYear the day-of-year
253  * @return the Thai Buddhist local date, not null
254  * @throws DateTimeException if unable to create the date
255  */
256 @Override
257 public ThaiBuddhistDate dateYearDay(int prolepticYear, int dayOfYear) {
258     return new ThaiBuddhistDate(LocalDate.ofYearDay(prolepticYear - YEARS_DIFFERENCE, dayOfYear)
259         ));
260 }
261
262 /**
263  * Obtains a local date in the Thai Buddhist calendar system from the epoch-day.
264  *
265  * @param epochDay the epoch day
266  * @return the Thai Buddhist local date, not null
267  * @throws DateTimeException if unable to create the date
268  */
269 @Override // override with covariant return type
270 public ThaiBuddhistDate dateEpochDay(long epochDay) {
271     return new ThaiBuddhistDate(LocalDate.ofEpochDay(epochDay));
272 }
273
274 @Override
275 public ThaiBuddhistDate dateNow() {
276     return dateNow(Clock.systemDefaultZone());
277 }
278
279 @Override
280 public ThaiBuddhistDate dateNow(ZoneId zone) {
281     return dateNow(Clock.system(zone));
282 }
283
284 @Override
285 public ThaiBuddhistDate dateNow(Clock clock) {
286     return date(LocalDate.now(clock));
287 }
288
289 @Override
290 public ThaiBuddhistDate date(TemporalAccessor temporal) {
291     if (temporal instanceof ThaiBuddhistDate) {
292         return (ThaiBuddhistDate) temporal;
293     }
294     return new ThaiBuddhistDate(LocalDate.from(temporal));
295 }

```

```

295
296 @Override
297 @SuppressWarnings("unchecked")
298 public ChronoLocalDateTime<ThaiBuddhistDate> localDateTime(TemporalAccessor temporal) {
299     return (ChronoLocalDateTime<ThaiBuddhistDate>)super.localDateTime(temporal);
300 }
301
302 @Override
303 @SuppressWarnings("unchecked")
304 public ChronoZonedDateTime<ThaiBuddhistDate> zonedDateTime(TemporalAccessor temporal) {
305     return (ChronoZonedDateTime<ThaiBuddhistDate>)super.zonedDateTime(temporal);
306 }
307
308 @Override
309 @SuppressWarnings("unchecked")
310 public ChronoZonedDateTime<ThaiBuddhistDate> zonedDateTime(Instant instant, ZoneId zone) {
311     return (ChronoZonedDateTime<ThaiBuddhistDate>)super.zonedDateTime(instant, zone);
312 }
313
314 //-----
315 /**
316  * Checks if the specified year is a leap year.
317  * <p>
318  * Thai Buddhist leap years occur exactly in line with ISO leap years.
319  * This method does not validate the year passed in, and only has a
320  * well-defined result for years in the supported range.
321  *
322  * @param prolepticYear the proleptic-year to check, not validated for range
323  * @return true if the year is a leap year
324  */
325 @Override
326 public boolean isLeapYear(long prolepticYear) {
327     return IsoChronology.INSTANCE.isLeapYear(prolepticYear - YEARS_DIFFERENCE);
328 }
329
330 @Override
331 public int prolepticYear(Era era, int yearOfEra) {
332     if (era instanceof ThaiBuddhistEra == false) {
333         throw new ClassCastException("Era must be BuddhistEra");
334     }
335     return (era == ThaiBuddhistEra.BE ? yearOfEra : 1 - yearOfEra);
336 }
337
338 @Override
339 public ThaiBuddhistEra eraOf(int eraValue) {
340     return ThaiBuddhistEra.of(eraValue);
341 }
342
343 @Override
344 public List<Era> eras() {
345     return Arrays.<Era>asList(ThaiBuddhistEra.values());
346 }
347

```

```

348 //-----
349 @Override
350 public ValueRange range(ChronoField field) {
351     switch (field) {
352         case PROLEPTIC_MONTH: {
353             ValueRange range = PROLEPTIC_MONTH.range();
354             return ValueRange.of(range.getMinimum() + YEARS_DIFFERENCE * 12L, range.getMaximum() + YEARS_DIFFERENCE * 12L);
355         }
356         case YEAR_OF_ERA: {
357             ValueRange range = YEAR.range();
358             return ValueRange.of(1, -(range.getMinimum() + YEARS_DIFFERENCE) + 1, range.getMaximum() + YEARS_DIFFERENCE);
359         }
360         case YEAR: {
361             ValueRange range = YEAR.range();
362             return ValueRange.of(range.getMinimum() + YEARS_DIFFERENCE, range.getMaximum() + YEARS_DIFFERENCE);
363         }
364     }
365     return field.range();
366 }
367
368 //-----
369 @Override // override for return type
370 public ThaiBuddhistDate resolveDate(Map<TemporalField, Long> fieldValues, ResolverStyle resolverStyle) {
371     return (ThaiBuddhistDate) super.resolveDate(fieldValues, resolverStyle);
372 }
373
374 //-----
375 /**
376  * Writes the Chronology using a
377  * <a href="../../serialized-form.html#java.time.chrono.Ser">dedicated serialized form</a>.
378  * @serialData
379  * <pre>
380  *   out.writeByte(1);    // identifies a Chronology
381  *   out.writeUTF(getId());
382  * </pre>
383  *
384  * @return the instance of {@code Ser}, not null
385  */
386 @Override
387 Object writeReplace() {
388     return super.writeReplace();
389 }
390
391 /**
392  * Defend against malicious streams.
393  *
394  * @param s the stream to read
395  * @throws InvalidObjectException always
396  */

```

```
397     private void readObject(ObjectInputStream s) throws InvalidObjectException {
398         throw new InvalidObjectException("Deserialization via serialization delegate");
399     }
400 }
```

5.25 ThaiBuddhistDate.java

Secuencia 95: java/time/chrono/ThaiBuddhistDate.java

```
1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
28 *
29 * All rights reserved.
30 *
31 * Redistribution and use in source and binary forms, with or without
32 * modification, are permitted provided that the following conditions are met:
33 *
34 * * Redistributions of source code must retain the above copyright notice,
35 *   this list of conditions and the following disclaimer.
36 *
37 * * Redistributions in binary form must reproduce the above copyright notice,
38 *   this list of conditions and the following disclaimer in the documentation
39 *   and/or other materials provided with the distribution.
40 *
41 * * Neither the name of JSR-310 nor the names of its contributors
42 *   may be used to endorse or promote products derived from this software
43 *   without specific prior written permission.
44 *
45 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
```

```

46  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56  */
57  package java.time.chrono;
58
59  import static java.time.chrono.ThaiBuddhistChronology.YEARS_DIFFERENCE;
60  import static java.time.temporal.ChronoField.DAY_OF_MONTH;
61  import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
62  import static java.time.temporal.ChronoField.YEAR;
63
64  import java.io.DataInput;
65  import java.io.DataOutput;
66  import java.io.IOException;
67  import java.io.InvalidObjectException;
68  import java.io.ObjectInputStream;
69  import java.io.Serializable;
70  import java.time.Clock;
71  import java.time.DateTimeException;
72  import java.time.LocalDate;
73  import java.time.LocalDateTime;
74  import java.time.Period;
75  import java.time.ZoneId;
76  import java.time.temporal.ChronoField;
77  import java.time.temporal.TemporalAccessor;
78  import java.time.temporal.TemporalAdjuster;
79  import java.time.temporal.TemporalAmount;
80  import java.time.temporal.TemporalField;
81  import java.time.temporal.TemporalQuery;
82  import java.time.temporal.TemporalUnit;
83  import java.time.temporal.UnsupportedTemporalTypeException;
84  import java.time.temporal.ValueRange;
85  import java.util.Objects;
86
87  /**
88   * A date in the Thai Buddhist calendar system.
89   * <p>
90   * This date operates using the {@linkplain ThaiBuddhistChronology Thai Buddhist calendar}.
91   * This calendar system is primarily used in Thailand.
92   * Dates are aligned such that {@code 2484-01-01 (Buddhist)} is {@code 1941-01-01 (ISO)}.
93   *
94   * <p>
95   * This is a <a href="{@docRoot}/java/lang/doc-files/ValueBased.html">value-based</a>
96   * class; use of identity-sensitive operations (including reference equality
97   * ({@code ==}), identity hash code, or synchronization) on instances of
98   * {@code ThaiBuddhistDate} may have unpredictable results and should be avoided.

```



```

99  * The {@code equals} method should be used for comparisons.
100 *
101 * @implSpec
102 * This class is immutable and thread-safe.
103 *
104 * @since 1.8
105 */
106 public final class ThaiBuddhistDate
107     extends ChronoLocalDateImpl<ThaiBuddhistDate>
108     implements ChronoLocalDate, Serializable {
109
110     /**
111      * Serialization version.
112      */
113     private static final long serialVersionUID = -8722293800195731463L;
114
115     /**
116      * The underlying date.
117      */
118     private final transient LocalDate isoDate;
119
120     //-----
121     /**
122      * Obtains the current {@code ThaiBuddhistDate} from the system clock in the default time-zone.
123      * <p>
124      * This will query the {@link Clock#systemDefaultZone() system clock} in the default
125      * time-zone to obtain the current date.
126      * <p>
127      * Using this method will prevent the ability to use an alternate clock for testing
128      * because the clock is hard-coded.
129      *
130      * @return the current date using the system clock and default time-zone, not null
131      */
132     public static ThaiBuddhistDate now() {
133         return now(Clock.systemDefaultZone());
134     }
135
136     /**
137      * Obtains the current {@code ThaiBuddhistDate} from the system clock in the specified time-
138      * zone.
139      * <p>
140      * This will query the {@link Clock#system(ZoneId) system clock} to obtain the current date.
141      * Specifying the time-zone avoids dependence on the default time-zone.
142      * <p>
143      * Using this method will prevent the ability to use an alternate clock for testing
144      * because the clock is hard-coded.
145      *
146      * @param zone the zone ID to use, not null
147      * @return the current date using the system clock, not null
148      */
149     public static ThaiBuddhistDate now(ZoneId zone) {
150         return now(Clock.system(zone));
151     }

```

```

151
152 /**
153  * Obtains the current {@code ThaiBuddhistDate} from the specified clock.
154  * <p>
155  * This will query the specified clock to obtain the current date - today.
156  * Using this method allows the use of an alternate clock for testing.
157  * The alternate clock may be introduced using {@linkplain Clock dependency injection}.
158  *
159  * @param clock the clock to use, not null
160  * @return the current date, not null
161  * @throws DateTimeException if the current date cannot be obtained
162  */
163 public static ThaiBuddhistDate now(Clock clock) {
164     return new ThaiBuddhistDate(LocalDate.now(clock));
165 }
166
167 /**
168  * Obtains a {@code ThaiBuddhistDate} representing a date in the Thai Buddhist calendar
169  * system from the proleptic-year, month-of-year and day-of-month fields.
170  * <p>
171  * This returns a {@code ThaiBuddhistDate} with the specified fields.
172  * The day must be valid for the year and month, otherwise an exception will be thrown.
173  *
174  * @param prolepticYear the Thai Buddhist proleptic-year
175  * @param month the Thai Buddhist month-of-year, from 1 to 12
176  * @param dayOfMonth the Thai Buddhist day-of-month, from 1 to 31
177  * @return the date in Thai Buddhist calendar system, not null
178  * @throws DateTimeException if the value of any field is out of range,
179  * or if the day-of-month is invalid for the month-year
180  */
181 public static ThaiBuddhistDate of(int prolepticYear, int month, int dayOfMonth) {
182     return new ThaiBuddhistDate(LocalDate.of(prolepticYear - YEARS_DIFFERENCE, month,
183         dayOfMonth));
184 }
185
186 /**
187  * Obtains a {@code ThaiBuddhistDate} from a temporal object.
188  * <p>
189  * This obtains a date in the Thai Buddhist calendar system based on the specified temporal.
190  * A {@code TemporalAccessor} represents an arbitrary set of date and time information,
191  * which this factory converts to an instance of {@code ThaiBuddhistDate}.
192  * <p>
193  * The conversion typically uses the {@link ChronoField#EPOCH_DAY EPOCH_DAY}
194  * field, which is standardized across calendar systems.
195  * <p>
196  * This method matches the signature of the functional interface {@link TemporalQuery}
197  * allowing it to be used as a query via method reference, {@code ThaiBuddhistDate::from}.
198  *
199  * @param temporal the temporal object to convert, not null
200  * @return the date in Thai Buddhist calendar system, not null
201  * @throws DateTimeException if unable to convert to a {@code ThaiBuddhistDate}
202  */
203 public static ThaiBuddhistDate from(TemporalAccessor temporal) {

```

```

203         return ThaiBuddhistChronology.INSTANCE.date(temporal);
204     }
205
206     //-----
207     /**
208      * Creates an instance from an ISO date.
209      *
210      * @param isoDate the standard local date, validated not null
211      */
212     ThaiBuddhistDate(LocalDate isoDate) {
213         Objects.requireNonNull(isoDate, "isoDate");
214         this.isoDate = isoDate;
215     }
216
217     //-----
218     /**
219      * Gets the chronology of this date, which is the Thai Buddhist calendar system.
220      * <p>
221      * The {@code Chronology} represents the calendar system in use.
222      * The era and other fields in {@link ChronoField} are defined by the chronology.
223      *
224      * @return the Thai Buddhist chronology, not null
225      */
226     @Override
227     public ThaiBuddhistChronology getChronology() {
228         return ThaiBuddhistChronology.INSTANCE;
229     }
230
231     /**
232      * Gets the era applicable at this date.
233      * <p>
234      * The Thai Buddhist calendar system has two eras, 'BE' and 'BEFORE_BE',
235      * defined by {@link ThaiBuddhistEra}.
236      *
237      * @return the era applicable at this date, not null
238      */
239     @Override
240     public ThaiBuddhistEra getEra() {
241         return (getProlepticYear() >= 1 ? ThaiBuddhistEra.BE : ThaiBuddhistEra.BEFORE_BE);
242     }
243
244     /**
245      * Returns the length of the month represented by this date.
246      * <p>
247      * This returns the length of the month in days.
248      * Month lengths match those of the ISO calendar system.
249      *
250      * @return the length of the month in days
251      */
252     @Override
253     public int lengthOfMonth() {
254         return isoDate.lengthOfMonth();
255     }

```

```

256
257 //-----
258 @Override
259 public ValueRange range(TemporalField field) {
260     if (field instanceof ChronoField) {
261         if (isSupported(field)) {
262             ChronoField f = (ChronoField) field;
263             switch (f) {
264                 case DAY_OF_MONTH:
265                 case DAY_OF_YEAR:
266                 case ALIGNED_WEEK_OF_MONTH:
267                     return isoDate.range(field);
268                 case YEAR_OF_ERA: {
269                     ValueRange range = YEAR.range();
270                     long max = (getProlepticYear() <= 0 ? -(range.getMinimum() +
271                                     YEARS_DIFFERENCE) + 1 : range.getMaximum() + YEARS_DIFFERENCE);
272                     return ValueRange.of(1, max);
273                 }
274             }
275             return getChronology().range(f);
276         }
277         throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
278     }
279     return field.rangeRefinedBy(this);
280 }
281
282 @Override
283 public long getLong(TemporalField field) {
284     if (field instanceof ChronoField) {
285         switch ((ChronoField) field) {
286             case PROLEPTIC_MONTH:
287                 return getProlepticMonth();
288             case YEAR_OF_ERA: {
289                 int prolepticYear = getProlepticYear();
290                 return (prolepticYear >= 1 ? prolepticYear : 1 - prolepticYear);
291             }
292             case YEAR:
293                 return getProlepticYear();
294             case ERA:
295                 return (getProlepticYear() >= 1 ? 1 : 0);
296         }
297         return isoDate.getLong(field);
298     }
299     return field.getFrom(this);
300 }
301
302 private long getProlepticMonth() {
303     return getProlepticYear() * 12L + isoDate.getMonthValue() - 1;
304 }
305
306 private int getProlepticYear() {
307     return isoDate.getYear() + YEARS_DIFFERENCE;
308 }

```

```

308
309 //-----
310 @Override
311 public ThaiBuddhistDate with(TemporalField field, long newValue) {
312     if (field instanceof ChronoField) {
313         ChronoField f = (ChronoField) field;
314         if (getLong(f) == newValue) {
315             return this;
316         }
317         switch (f) {
318             case PROLEPTIC_MONTH:
319                 getChronology().range(f).checkValidValue(newValue, f);
320                 return plusMonths(newValue - getProlepticMonth());
321             case YEAR_OF_ERA:
322             case YEAR:
323             case ERA: {
324                 int nvalue = getChronology().range(f).checkValidIntValue(newValue, f);
325                 switch (f) {
326                     case YEAR_OF_ERA:
327                         return with(isoDate.withYear((getProlepticYear() >= 1 ? nvalue : 1 -
328                             nvalue) - YEARS_DIFFERENCE));
329                     case YEAR:
330                         return with(isoDate.withYear(nvalue - YEARS_DIFFERENCE));
331                     case ERA:
332                         return with(isoDate.withYear((1 - getProlepticYear()) -
333                             YEARS_DIFFERENCE));
334                 }
335             }
336         }
337         return with(isoDate.with(field, newValue));
338     }
339     return super.with(field, newValue);
340 }
341
342 /**
343  * {@inheritDoc}
344  * @throws DateTimeException {@inheritDoc}
345  * @throws ArithmeticException {@inheritDoc}
346  */
347 @Override
348 public ThaiBuddhistDate with(TemporalAdjuster adjuster) {
349     return super.with(adjuster);
350 }
351
352 /**
353  * {@inheritDoc}
354  * @throws DateTimeException {@inheritDoc}
355  * @throws ArithmeticException {@inheritDoc}
356  */
357 @Override
358 public ThaiBuddhistDate plus(TemporalAmount amount) {
359     return super.plus(amount);
360 }

```

```
359
360 /**
361  * {@inheritDoc}
362  * @throws DateTimeException {@inheritDoc}
363  * @throws ArithmeticException {@inheritDoc}
364  */
365 @Override
366 public ThaiBuddhistDate minus(TemporalAmount amount) {
367     return super.minus(amount);
368 }
369
370 //-----
371 @Override
372 ThaiBuddhistDate plusYears(long years) {
373     return with(isoDate.plusYears(years));
374 }
375
376 @Override
377 ThaiBuddhistDate plusMonths(long months) {
378     return with(isoDate.plusMonths(months));
379 }
380
381 @Override
382 ThaiBuddhistDate plusWeeks(long weeksToAdd) {
383     return super.plusWeeks(weeksToAdd);
384 }
385
386 @Override
387 ThaiBuddhistDate plusDays(long days) {
388     return with(isoDate.plusDays(days));
389 }
390
391 @Override
392 public ThaiBuddhistDate plus(long amountToAdd, TemporalUnit unit) {
393     return super.plus(amountToAdd, unit);
394 }
395
396 @Override
397 public ThaiBuddhistDate minus(long amountToAdd, TemporalUnit unit) {
398     return super.minus(amountToAdd, unit);
399 }
400
401 @Override
402 ThaiBuddhistDate minusYears(long yearsToSubtract) {
403     return super.minusYears(yearsToSubtract);
404 }
405
406 @Override
407 ThaiBuddhistDate minusMonths(long monthsToSubtract) {
408     return super.minusMonths(monthsToSubtract);
409 }
410
411 @Override
```

```

412     ThaiBuddhistDate minusWeeks(long weeksToSubtract) {
413         return super.minusWeeks(weeksToSubtract);
414     }
415
416     @Override
417     ThaiBuddhistDate minusDays(long daysToSubtract) {
418         return super.minusDays(daysToSubtract);
419     }
420
421     private ThaiBuddhistDate with(LocalDate newDate) {
422         return (newDate.equals(isoDate) ? this : new ThaiBuddhistDate(newDate));
423     }
424
425     @Override          // for javadoc and covariant return type
426     @SuppressWarnings("unchecked")
427     public final ChronoLocalDateTime<ThaiBuddhistDate> atTime(LocalTime localTime) {
428         return (ChronoLocalDateTime<ThaiBuddhistDate>) super.atTime(localTime);
429     }
430
431     @Override
432     public ChronoPeriod until(ChronoLocalDate endDate) {
433         Period period = isoDate.until(endDate);
434         return getChronology().period(period.getYears(), period.getMonths(), period.getDays());
435     }
436
437     @Override // override for performance
438     public long toEpochDay() {
439         return isoDate.toEpochDay();
440     }
441
442     //-----
443     /**
444      * Compares this date to another date, including the chronology.
445      * <p>
446      * Compares this {@code ThaiBuddhistDate} with another ensuring that the date is the same.
447      * <p>
448      * Only objects of type {@code ThaiBuddhistDate} are compared, other types return false.
449      * To compare the dates of two {@code TemporalAccessor} instances, including dates
450      * in two different chronologies, use {@link ChronoField#EPOCH_DAY} as a comparator.
451      *
452      * @param obj the object to check, null returns false
453      * @return true if this is equal to the other date
454      */
455     @Override // override for performance
456     public boolean equals(Object obj) {
457         if (this == obj) {
458             return true;
459         }
460         if (obj instanceof ThaiBuddhistDate) {
461             ThaiBuddhistDate otherDate = (ThaiBuddhistDate) obj;
462             return this.isoDate.equals(otherDate.isoDate);
463         }
464         return false;

```

```

465     }
466
467     /**
468      * A hash code for this date.
469      *
470      * @return a suitable hash code based only on the Chronology and the date
471      */
472     @Override // override for performance
473     public int hashCode() {
474         return getChronology().getId().hashCode() ^ isoDate.hashCode();
475     }
476
477     //-----
478     /**
479      * Defend against malicious streams.
480      *
481      * @param s the stream to read
482      * @throws InvalidObjectException always
483      */
484     private void readObject(ObjectInputStream s) throws InvalidObjectException {
485         throw new InvalidObjectException("Deserialization via serialization delegate");
486     }
487
488     /**
489      * Writes the object using a
490      * <a href="../../../../serialized-form.html#java.time.chrono.Ser">dedicated serialized form</a>.
491      * @serialData
492      * <pre>
493      *   out.writeByte(10);           // identifies a ThaiBuddhistDate
494      *   out.writeInt(get(YEAR));
495      *   out.writeByte(get(MONTH_OF_YEAR));
496      *   out.writeByte(get(DAY_OF_MONTH));
497      * </pre>
498      *
499      * @return the instance of {@code Ser}, not null
500      */
501     private Object writeReplace() {
502         return new Ser(Ser.THAIBUDDHIST_DATE_TYPE, this);
503     }
504
505     void writeExternal(DataOutput out) throws IOException {
506         // ThaiBuddhistChronology is implicit in the THAIBUDDHIST_DATE_TYPE
507         out.writeInt(this.get(YEAR));
508         out.writeByte(this.get(MONTH_OF_YEAR));
509         out.writeByte(this.get(DAY_OF_MONTH));
510     }
511
512     static ThaiBuddhistDate readExternal(DataInput in) throws IOException {
513         int year = in.readInt();
514         int month = in.readByte();
515         int dayOfMonth = in.readByte();
516         return ThaiBuddhistChronology.INSTANCE.date(year, month, dayOfMonth);
517     }

```



```
518  
519 }
```

5.26 ThaiBuddhistEra.java

Secuencia 96: java/time/chrono/ThaiBuddhistEra.java

```
1  /*  
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.  
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.  
4  *  
5  *  
6  *  
7  *  
8  *  
9  *  
10 *  
11 *  
12 *  
13 *  
14 *  
15 *  
16 *  
17 *  
18 *  
19 *  
20 *  
21 *  
22 *  
23 *  
24 */  
25  
26 /*  
27 *  
28 *  
29 *  
30 *  
31 *  
32 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos  
33 *  
34 * All rights reserved.  
35 *  
36 * Redistribution and use in source and binary forms, with or without  
37 * modification, are permitted provided that the following conditions are met:  
38 *  
39 * * Redistributions of source code must retain the above copyright notice,  
40 *   this list of conditions and the following disclaimer.  
41 *  
42 * * Redistributions in binary form must reproduce the above copyright notice,  
43 *   this list of conditions and the following disclaimer in the documentation  
44 *   and/or other materials provided with the distribution.  
45 *  
46 * * Neither the name of JSR-310 nor the names of its contributors  
47 *   may be used to endorse or promote products derived from this software
```

```

48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.chrono;
63
64  import java.time.DateTimeException;
65
66  /**
67   * An era in the Thai Buddhist calendar system.
68   * <p>
69   * The Thai Buddhist calendar system has two eras.
70   * The current era, for years from 1 onwards, is known as the 'Buddhist' era.
71   * All previous years, zero or earlier in the proleptic count or one and greater
72   * in the year-of-era count, are part of the 'Before Buddhist' era.
73   *
74   * <table summary="Buddhist years and eras" cellpadding="2" cellspacing="3" border="0" >
75   * <thead>
76   * <tr class="tableSubHeadingColor">
77   * <th class="colFirst" align="left">year-of-era</th>
78   * <th class="colFirst" align="left">era</th>
79   * <th class="colFirst" align="left">proleptic-year</th>
80   * <th class="colLast" align="left">ISO proleptic-year</th>
81   * </tr>
82   * </thead>
83   * <tbody>
84   * <tr class="rowColor">
85   * <td>2</td><td>BE</td><td>2</td><td>-542</td>
86   * </tr>
87   * <tr class="altColor">
88   * <td>1</td><td>BE</td><td>1</td><td>-543</td>
89   * </tr>
90   * <tr class="rowColor">
91   * <td>1</td><td>BEFORE_BE</td><td>0</td><td>-544</td>
92   * </tr>
93   * <tr class="altColor">
94   * <td>2</td><td>BEFORE_BE</td><td>-1</td><td>-545</td>
95   * </tr>
96   * </tbody>
97   * </table>
98   * <p>
99   * <b>Do not use {@code ordinal()} to obtain the numeric representation of {@code ThaiBuddhistEra}.
100  * Use {@code getValue()} instead.</b>

```

```

101  *
102  * @implSpec
103  * This is an immutable and thread-safe enum.
104  *
105  * @since 1.8
106  */
107  public enum ThaiBuddhistEra implements Era {
108
109      /**
110       * The singleton instance for the era before the current one, 'Before Buddhist Era',
111       * which has the numeric value 0.
112       */
113      BEFORE_BE,
114
115      /**
116       * The singleton instance for the current era, 'Buddhist Era',
117       * which has the numeric value 1.
118       */
119      BE;
120
121      //-----
122      /**
123       * Obtains an instance of {@code ThaiBuddhistEra} from an {@code int} value.
124       * <p>
125       * {@code ThaiBuddhistEra} is an enum representing the Thai Buddhist eras of BEFORE_BE/BE.
126       * This factory allows the enum to be obtained from the {@code int} value.
127       *
128       * @param thaiBuddhistEra the era to represent, from 0 to 1
129       * @return the BuddhistEra singleton, never null
130       * @throws DateTimeException if the era is invalid
131       */
132      public static ThaiBuddhistEra of(int thaiBuddhistEra) {
133          switch (thaiBuddhistEra) {
134              case 0:
135                  return BEFORE_BE;
136              case 1:
137                  return BE;
138              default:
139                  throw new DateTimeException("Invalid era: " + thaiBuddhistEra);
140          }
141      }
142
143      //-----
144      /**
145       * Gets the numeric era {@code int} value.
146       * <p>
147       * The era BEFORE_BE has the value 0, while the era BE has the value 1.
148       *
149       * @return the era value, from 0 (BEFORE_BE) to 1 (BE)
150       */
151      @Override
152      public int getValue() {
153          return ordinal();
154      }
155  }

```

```
154  
155 }
```

5.27 package-info.java

Secuencia 97: java/time/chrono/package-info.java

```
1  /*  
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.  
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.  
4  *  
5  *  
6  *  
7  *  
8  *  
9  *  
10 *  
11 *  
12 *  
13 *  
14 *  
15 *  
16 *  
17 *  
18 *  
19 *  
20 *  
21 *  
22 *  
23 *  
24 */  
25  
26 /*  
27 *  
28 *  
29 *  
30 *  
31 *  
32 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos  
33 *  
34 * All rights reserved.  
35 *  
36 * Redistribution and use in source and binary forms, with or without  
37 * modification, are permitted provided that the following conditions are met:  
38 *  
39 * * Redistributions of source code must retain the above copyright notice,  
40 *   this list of conditions and the following disclaimer.  
41 *  
42 * * Redistributions in binary form must reproduce the above copyright notice,  
43 *   this list of conditions and the following disclaimer in the documentation  
44 *   and/or other materials provided with the distribution.  
45 *  
46 * * Neither the name of JSR-310 nor the names of its contributors  
47 *   may be used to endorse or promote products derived from this software
```

```

48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
62
63 /**
64 * <p>
65 * Generic API for calendar systems other than the default ISO.
66 * </p>
67 * <p>
68 * The main API is based around the calendar system defined in ISO-8601.
69 * However, there are other calendar systems, and this package provides basic support for them.
70 * The alternate calendars are provided in the {@link java.time.chrono} package.
71 * </p>
72 * <p>
73 * A calendar system is defined by the {@link java.time.chrono.Chronology} interface,
74 * while a date in a calendar system is defined by the {@link java.time.chrono.ChronoLocalDate}
75 *   interface.
76 * </p>
77 * <p>
78 * It is intended that applications use the main API whenever possible, including code to read and
79 *   write
80 * from a persistent data store, such as a database, and to send dates and times across a network.
81 * The "chrono" classes are then used at the user interface level to deal with localized input/
82 *   output.
83 * </p>
84 * See {@link java.time.chrono.ChronoLocalDate ChronoLocalDate}
85 * for a full discussion of the issues.
86 * </p>
87 * <p>
88 * Using non-ISO calendar systems in an application introduces significant extra complexity.
89 * Ensure that the warnings and recommendations in {@code ChronoLocalDate} have been read before
90 * working with the "chrono" interfaces.
91 * </p>
92 * <p>
93 * The supported calendar systems includes:
94 * </p>
95 * <ul>
96 * <li>{@link java.time.chrono.HijrahChronology Hijrah calendar}</li>
97 * <li>{@link java.time.chrono.JapaneseChronology Japanese calendar}</li>
98 * <li>{@link java.time.chrono.MinguoChronology Minguo calendar}</li>
99 * <li>{@link java.time.chrono.ThaiBuddhistChronology Thai Buddhist calendar}</li>
100 * </ul>
101 *

```

```

98 * <h3>Example</h3>
99 * <p>
100 * This example lists today's date for all of the available calendars.
101 * </p>
102 * <pre>
103 * // Enumerate the list of available calendars and print today's date for each.
104 *     Set<Chronology> chronos = Chronology.getAvailableChronologies();
105 *     for (Chronology chrono : chronos) {
106 *         ChronoLocalDate date = chrono.dateNow();
107 *         System.out.printf("    %20s: %s%n", chrono.getId(), date.toString());
108 *     }
109 * </pre>
110 *
111 * <p>
112 * This example creates and uses a date in a named non-ISO calendar system.
113 * </p>
114 * <pre>
115 * // Print the Thai Buddhist date
116 *     ChronoLocalDate now1 = Chronology.of("ThaiBuddhist").dateNow();
117 *     int day = now1.get(ChronoField.DAY_OF_MONTH);
118 *     int dow = now1.get(ChronoField.DAY_OF_WEEK);
119 *     int month = now1.get(ChronoField.MONTH_OF_YEAR);
120 *     int year = now1.get(ChronoField.YEAR);
121 *     System.out.printf("    Today is %s %s %d-%s-%d%n", now1.getChronology().getId(),
122 *         dow, day, month, year);
123 * // Print today's date and the last day of the year for the Thai Buddhist Calendar.
124 *     ChronoLocalDate first = now1
125 *         .with(ChronoField.DAY_OF_MONTH, 1)
126 *         .with(ChronoField.MONTH_OF_YEAR, 1);
127 *     ChronoLocalDate last = first
128 *         .plus(1, ChronoUnit.YEARS)
129 *         .minus(1, ChronoUnit.DAYS);
130 *     System.out.printf("    %s: 1st of year: %s; end of year: %s%n", last.getChronology().getId()
131 *         ,
132 *         first, last);
133 * </pre>
134 *
135 * <p>
136 * This example creates and uses a date in a specific ThaiBuddhist calendar system.
137 * </p>
138 * <pre>
139 * // Print the Thai Buddhist date
140 *     ThaiBuddhistDate now1 = ThaiBuddhistDate.now();
141 *     int day = now1.get(ChronoField.DAY_OF_MONTH);
142 *     int dow = now1.get(ChronoField.DAY_OF_WEEK);
143 *     int month = now1.get(ChronoField.MONTH_OF_YEAR);
144 *     int year = now1.get(ChronoField.YEAR);
145 *     System.out.printf("    Today is %s %s %d-%s-%d%n", now1.getChronology().getId(),
146 *         dow, day, month, year);
147 *
148 * // Print today's date and the last day of the year for the Thai Buddhist Calendar.
149 *     ThaiBuddhistDate first = now1
150 *         .with(ChronoField.DAY_OF_MONTH, 1)

```

```

150 *         .with(ChronoField.MONTH_OF_YEAR, 1);
151 *         ThaiBuddhistDate last = first
152 *             .plus(1, ChronoUnit.YEARS)
153 *             .minus(1, ChronoUnit.DAYS);
154 *         System.out.printf("  %s: 1st of year: %s; end of year: %s%n", last.getChronology().getId()
155 *             ,
156 *             first, last);
157 *     </pre>
158 * <h3>Package specification</h3>
159 * <p>
160 * Unless otherwise noted, passing a null argument to a constructor or method in any class or
161 *     interface
162 * in this package will cause a {@link java.lang.NullPointerException NullPointerException} to be
163 *     thrown.
164 * The Javadoc "@param" definition is used to summarise the null-behavior.
165 * The "@throws {@link java.lang.NullPointerException}" is not explicitly documented in each method
166 * .
167 * </p>
168 * <p>
169 * All calculations should check for numeric overflow and throw either an {@link java.lang.
170 *     ArithmeticException}
171 * or a {@link java.time.DateTimeException}.
172 * </p>
173 * @since JDK1.8
174 */
175 package java.time.chrono;

```

6 Time Format (java.time.format package)

6.1 DateTimeFormatter.java

Secuencia 98: java/time/format/DateTimeFormatter.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *

```

```
21  *
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2008-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.format;
63
64  import static java.time.temporal.ChronoField.DAY_OF_MONTH;
65  import static java.time.temporal.ChronoField.DAY_OF_WEEK;
66  import static java.time.temporal.ChronoField.DAY_OF_YEAR;
67  import static java.time.temporal.ChronoField.HOUR_OF_DAY;
68  import static java.time.temporal.ChronoField.MINUTE_OF_HOUR;
69  import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
70  import static java.time.temporal.ChronoField.NANO_OF_SECOND;
71  import static java.time.temporal.ChronoField.SECOND_OF_MINUTE;
72  import static java.time.temporal.ChronoField.YEAR;
73
```



```
74 import java.io.IOException;
75 import java.text.FieldPosition;
76 import java.text.Format;
77 import java.text.ParseException;
78 import java.text.ParsePosition;
79 import java.time.DateTimeException;
80 import java.time.Period;
81 import java.time.ZoneId;
82 import java.time.ZoneOffset;
83 import java.time.chrono.Chronology;
84 import java.time.chrono.IsoChronology;
85 import java.time.format.DateTimeFormatterBuilder.CompositePrinterParser;
86 import java.time.temporal.ChronoField;
87 import java.time.temporal.IsoFields;
88 import java.time.temporal.TemporalAccessor;
89 import java.time.temporal.TemporalField;
90 import java.time.temporal.TemporalQuery;
91 import java.util.Arrays;
92 import java.util.Collections;
93 import java.util.HashMap;
94 import java.util.HashSet;
95 import java.util.Locale;
96 import java.util.Map;
97 import java.util.Objects;
98 import java.util.Set;
99
100 /**
101  * Formatter for printing and parsing date-time objects.
102  * <p>
103  * This class provides the main application entry point for printing and parsing
104  * and provides common implementations of {@code DateTimeFormatter}:
105  * <ul>
106  * <li>Using predefined constants, such as {@link #ISO_LOCAL_DATE}</li>
107  * <li>Using pattern letters, such as {@code uuuu-MMM-dd}</li>
108  * <li>Using localized styles, such as {@code long} or {@code medium}</li>
109  * </ul>
110  * <p>
111  * More complex formatters are provided by
112  * {@link DateTimeFormatterBuilder DateTimeFormatterBuilder}.
113  *
114  * <p>
115  * The main date-time classes provide two methods - one for formatting,
116  * {@code format(DateTimeFormatter formatter)}, and one for parsing,
117  * {@code parse(CharSequence text, DateTimeFormatter formatter)}.
118  * <p>For example:
119  * <blockquote><pre>
120  *   LocalDate date = LocalDate.now();
121  *   String text = date.format(formatter);
122  *   LocalDate parsedDate = LocalDate.parse(text, formatter);
123  * </pre></blockquote>
124  * <p>
125  * In addition to the format, formatters can be created with desired Locale,
126  * Chronology, ZoneId, and DecimalStyle.
```

```

127 * <p>
128 * The {@link #withLocale withLocale} method returns a new formatter that
129 * overrides the locale. The locale affects some aspects of formatting and
130 * parsing. For example, the {@link #ofLocalizedDate ofLocalizedDate} provides a
131 * formatter that uses the locale specific date format.
132 * <p>
133 * The {@link #withChronology withChronology} method returns a new formatter
134 * that overrides the chronology. If overridden, the date-time value is
135 * converted to the chronology before formatting. During parsing the date-time
136 * value is converted to the chronology before it is returned.
137 * <p>
138 * The {@link #withZone withZone} method returns a new formatter that overrides
139 * the zone. If overridden, the date-time value is converted to a ZonedDateTime
140 * with the requested ZoneId before formatting. During parsing the ZoneId is
141 * applied before the value is returned.
142 * <p>
143 * The {@link #withDecimalStyle withDecimalStyle} method returns a new formatter that
144 * overrides the {@link DecimalStyle}. The DecimalStyle symbols are used for
145 * formatting and parsing.
146 * <p>
147 * Some applications may need to use the older {@link Format java.text.Format}
148 * class for formatting. The {@link #toFormat()} method returns an
149 * implementation of {@code java.text.Format}.
150 *
151 * <h3 id="predefined">Predefined Formatters</h3>
152 * <table summary="Predefined Formatters" cellpadding="2" cellspacing="3" border="0" >
153 * <thead>
154 * <tr class="tableSubHeadingColor">
155 * <th class="colFirst" align="left">Formatter</th>
156 * <th class="colFirst" align="left">Description</th>
157 * <th class="colLast" align="left">Example</th>
158 * </tr>
159 * </thead>
160 * <tbody>
161 * <tr class="rowColor">
162 * <td>{@link #ofLocalizedDate ofLocalizedDate(dateStyle)} </td>
163 * <td>Formatter with date style from the locale </td>
164 * <td>'2011-12-03'</td>
165 * </tr>
166 * <tr class="altColor">
167 * <td>{@link #ofLocalizedTime ofLocalizedTime(timeStyle)} </td>
168 * <td>Formatter with time style from the locale </td>
169 * <td>'10:15:30'</td>
170 * </tr>
171 * <tr class="rowColor">
172 * <td>{@link #ofLocalizedDateTime ofLocalizedDateTime(dateTimeStyle)} </td>
173 * <td>Formatter with a style for date and time from the locale</td>
174 * <td>'3 Jun 2008 11:05:30'</td>
175 * </tr>
176 * <tr class="altColor">
177 * <td>{@link #ofLocalizedDateTime ofLocalizedDateTime(dateStyle,timeStyle)}
178 * </td>
179 * <td>Formatter with date and time styles from the locale </td>

```

```

180 * <td> '3 Jun 2008 11:05'</td>
181 * </tr>
182 * <tr class="rowColor">
183 * <td> {@link #BASIC_ISO_DATE}</td>
184 * <td>Basic ISO date </td> <td>'20111203'</td>
185 * </tr>
186 * <tr class="altColor">
187 * <td> {@link #ISO_LOCAL_DATE}</td>
188 * <td> ISO Local Date </td>
189 * <td>'2011-12-03'</td>
190 * </tr>
191 * <tr class="rowColor">
192 * <td> {@link #ISO_OFFSET_DATE}</td>
193 * <td> ISO Date with offset </td>
194 * <td>'2011-12-03+01:00'</td>
195 * </tr>
196 * <tr class="altColor">
197 * <td> {@link #ISO_DATE}</td>
198 * <td> ISO Date with or without offset </td>
199 * <td> '2011-12-03+01:00'; '2011-12-03'</td>
200 * </tr>
201 * <tr class="rowColor">
202 * <td> {@link #ISO_LOCAL_TIME}</td>
203 * <td> Time without offset </td>
204 * <td>'10:15:30'</td>
205 * </tr>
206 * <tr class="altColor">
207 * <td> {@link #ISO_OFFSET_TIME}</td>
208 * <td> Time with offset </td>
209 * <td>'10:15:30+01:00'</td>
210 * </tr>
211 * <tr class="rowColor">
212 * <td> {@link #ISO_TIME}</td>
213 * <td> Time with or without offset </td>
214 * <td>'10:15:30+01:00'; '10:15:30'</td>
215 * </tr>
216 * <tr class="altColor">
217 * <td> {@link #ISO_LOCAL_DATE_TIME}</td>
218 * <td> ISO Local Date and Time </td>
219 * <td>'2011-12-03T10:15:30'</td>
220 * </tr>
221 * <tr class="rowColor">
222 * <td> {@link #ISO_OFFSET_DATE_TIME}</td>
223 * <td> Date Time with Offset
224 * </td><td>2011-12-03T10:15:30+01:00'</td>
225 * </tr>
226 * <tr class="altColor">
227 * <td> {@link #ISO_ZONED_DATE_TIME}</td>
228 * <td> Zoned Date Time </td>
229 * <td>'2011-12-03T10:15:30+01:00[Europe/Paris]'</td>
230 * </tr>
231 * <tr class="rowColor">
232 * <td> {@link #ISO_DATE_TIME}</td>

```

```

233 * <td> Date and time with ZoneId </td>
234 * <td>'2011-12-03T10:15:30+01:00 [Europe/Paris]' </td>
235 * </tr>
236 * <tr class="altColor">
237 * <td> {@link #ISO_ORDINAL_DATE}</td>
238 * <td> Year and day of year </td>
239 * <td>'2012-337'</td>
240 * </tr>
241 * <tr class="rowColor">
242 * <td> {@link #ISO_WEEK_DATE}</td>
243 * <td> Year and Week </td>
244 * <td>2012-W48-6'</td></tr>
245 * <tr class="altColor">
246 * <td> {@link #ISO_INSTANT}</td>
247 * <td> Date and Time of an Instant </td>
248 * <td>'2011-12-03T10:15:30Z' </td>
249 * </tr>
250 * <tr class="rowColor">
251 * <td> {@link #RFC_1123_DATE_TIME}</td>
252 * <td> RFC 1123 / RFC 822 </td>
253 * <td>'Tue, 3 Jun 2008 11:05:30 GMT'</td>
254 * </tr>
255 * </tbody>
256 * </table>
257 *
258 * <h3 id="patterns">Patterns for Formatting and Parsing</h3>
259 * Patterns are based on a simple sequence of letters and symbols.
260 * A pattern is used to create a Formatter using the
261 * {@link #ofPattern(String)} and {@link #ofPattern(String, Locale)} methods.
262 * For example,
263 * {@code "d MMM uuuu"} will format 2011-12-03 as '3&nbsp;Dec&nbsp;2011'.
264 * A formatter created from a pattern can be used as many times as necessary,
265 * it is immutable and is thread-safe.
266 * <p>
267 * For example:
268 * <blockquote><pre>
269 *   LocalDate date = LocalDate.now();
270 *   DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy MM dd");
271 *   String text = date.format(formatter);
272 *   LocalDate parsedDate = LocalDate.parse(text, formatter);
273 * </pre></blockquote>
274 * <p>
275 * All letters 'A' to 'Z' and 'a' to 'z' are reserved as pattern letters. The
276 * following pattern letters are defined:
277 * <pre>
278 *   Symbol   Meaning                               Presentation   Examples
279 *   -----   -----                               -
280 *   G         era                                   text           AD; Anno Domini; A
281 *   u         year                                   year           2004; 04
282 *   y         year-of-era                           year           2004; 04
283 *   D         day-of-year                           number          189
284 *   M/L       month-of-year                         number/text     7; 07; Jul; July; J
285 *   d         day-of-month                           number          10

```

```

286 *
287 * Q/q    quarter-of-year      number/text    3; 03; Q3; 3rd quarter
288 * Y      week-based-year      year           1996; 96
289 * w      week-of-week-based-year number          27
290 * W      week-of-month        number          4
291 * E      day-of-week          text            Tue; Tuesday; T
292 * e/c    localized day-of-week number/text        2; 02; Tue; Tuesday; T
293 * F      week-of-month        number          3
294 *
295 * a      am-pm-of-day          text            PM
296 * h      clock-hour-of-am-pm (1-12) number          12
297 * K      hour-of-am-pm (0-11)  number          0
298 * k      clock-hour-of-am-pm (1-24) number          0
299 *
300 * H      hour-of-day (0-23)    number          0
301 * m      minute-of-hour       number          30
302 * s      second-of-minute     number          55
303 * S      fraction-of-second   fraction         978
304 * A      milli-of-day         number          1234
305 * n      nano-of-second       number          987654321
306 * N      nano-of-day          number          1234000000
307 *
308 * V      time-zone ID          zone-id         America/Los_Angeles; Z; -08:30
309 * z      time-zone name        zone-name       Pacific Standard Time; PST
310 * O      localized zone-offset offset-0         GMT+8; GMT+08:00; UTC-08:00;
311 * X      zone-offset 'Z' for zero offset-X         Z; -08; -0830; -08:30; -083015;
312 * x      zone-offset          offset-x         -08:30:15; +0000; -08; -0830; -08:30; -083015;
313 * Z      zone-offset          offset-Z         -08:30:15; +0000; -0800; -08:00;
314 *
315 * p      pad next              pad modifier    1
316 *
317 * '      escape for text       delimiter
318 * ''     single quote         literal        '
319 * [      optional section start
320 * ]      optional section end
321 * #      reserved for future use
322 * {      reserved for future use
323 * }      reserved for future use
324 * </pre>
325 * <p>
326 * The count of pattern letters determines the format.
327 * <p>
328 * <b>Text</b>: The text style is determined based on the number of pattern
329 * letters used. Less than 4 pattern letters will use the
330 * {@link TextStyle#SHORT short form}. Exactly 4 pattern letters will use the
331 * {@link TextStyle#FULL full form}. Exactly 5 pattern letters will use the
332 * {@link TextStyle#NARROW narrow form}.
333 * Pattern letters 'L', 'c', and 'q' specify the stand-alone form of the text styles.
334 * <p>
335 * <b>Number</b>: If the count of letters is one, then the value is output using
336 * the minimum number of digits and without padding. Otherwise, the count of digits

```

```

337 * is used as the width of the output field, with the value zero-padded as necessary.
338 * The following pattern letters have constraints on the count of letters.
339 * Only one letter of 'c' and 'F' can be specified.
340 * Up to two letters of 'd', 'H', 'h', 'K', 'k', 'm', and 's' can be specified.
341 * Up to three letters of 'D' can be specified.
342 * <p>
343 * <b>Number/Text</b>: If the count of pattern letters is 3 or greater, use the
344 * Text rules above. Otherwise use the Number rules above.
345 * <p>
346 * <b>Fraction</b>: Outputs the nano-of-second field as a fraction-of-second.
347 * The nano-of-second value has nine digits, thus the count of pattern letters
348 * is from 1 to 9. If it is less than 9, then the nano-of-second value is
349 * truncated, with only the most significant digits being output.
350 * <p>
351 * <b>Year</b>: The count of letters determines the minimum field width below
352 * which padding is used. If the count of letters is two, then a
353 * {@link DateTimeFormatterBuilder#appendValueReduced reduced} two digit form is
354 * used. For printing, this outputs the rightmost two digits. For parsing, this
355 * will parse using the base value of 2000, resulting in a year within the range
356 * 2000 to 2099 inclusive. If the count of letters is less than four (but not
357 * two), then the sign is only output for negative years as per
358 * {@link SignStyle#NORMAL}. Otherwise, the sign is output if the pad width is
359 * exceeded, as per {@link SignStyle#EXCEEDS_PAD}.
360 * <p>
361 * <b>ZoneId</b>: This outputs the time-zone ID, such as 'Europe/Paris'. If the
362 * count of letters is two, then the time-zone ID is output. Any other count of
363 * letters throws {@code IllegalArgumentException}.
364 * <p>
365 * <b>Zone names</b>: This outputs the display name of the time-zone ID. If the
366 * count of letters is one, two or three, then the short name is output. If the
367 * count of letters is four, then the full name is output. Five or more letters
368 * throws {@code IllegalArgumentException}.
369 * <p>
370 * <b>Offset X and x</b>: This formats the offset based on the number of pattern
371 * letters. One letter outputs just the hour, such as '+01', unless the minute
372 * is non-zero in which case the minute is also output, such as '+0130'. Two
373 * letters outputs the hour and minute, without a colon, such as '+0130'. Three
374 * letters outputs the hour and minute, with a colon, such as '+01:30'. Four
375 * letters outputs the hour and minute and optional second, without a colon,
376 * such as '+013015'. Five letters outputs the hour and minute and optional
377 * second, with a colon, such as '+01:30:15'. Six or more letters throws
378 * {@code IllegalArgumentException}. Pattern letter 'X' (upper case) will output
379 * 'Z' when the offset to be output would be zero, whereas pattern letter 'x'
380 * (lower case) will output '+00', '+0000', or '+00:00'.
381 * <p>
382 * <b>Offset 0</b>: This formats the localized offset based on the number of
383 * pattern letters. One letter outputs the {@linkplain TextStyle#SHORT short}
384 * form of the localized offset, which is localized offset text, such as 'GMT',
385 * with hour without leading zero, optional 2-digit minute and second if
386 * non-zero, and colon, for example 'GMT+8'. Four letters outputs the
387 * {@linkplain TextStyle#FULL full} form, which is localized offset text,
388 * such as 'GMT', with 2-digit hour and minute field, optional second field
389 * if non-zero, and colon, for example 'GMT+08:00'. Any other count of letters

```

```

390 * throws {@code IllegalArgumentException}.
391 * <p>
392 * <b>Offset Z</b>: This formats the offset based on the number of pattern
393 * letters. One, two or three letters outputs the hour and minute, without a
394 * colon, such as '+0130'. The output will be '+0000' when the offset is zero.
395 * Four letters outputs the {@linkplain TextStyle#FULL full} form of localized
396 * offset, equivalent to four letters of Offset-0. The output will be the
397 * corresponding localized offset text if the offset is zero. Five
398 * letters outputs the hour, minute, with optional second if non-zero, with
399 * colon. It outputs 'Z' if the offset is zero.
400 * Six or more letters throws {@code IllegalArgumentException}.
401 * <p>
402 * <b>Optional section</b>: The optional section markers work exactly like
403 * calling {@link DateTimeFormatterBuilder#optionalStart()} and
404 * {@link DateTimeFormatterBuilder#optionalEnd()}.
405 * <p>
406 * <b>Pad modifier</b>: Modifies the pattern that immediately follows to be
407 * padded with spaces. The pad width is determined by the number of pattern
408 * letters. This is the same as calling
409 * {@link DateTimeFormatterBuilder#padNext(int)}.
410 * <p>
411 * For example, 'ppH' outputs the hour-of-day padded on the left with spaces to
412 * a width of 2.
413 * <p>
414 * Any unrecognized letter is an error. Any non-letter character, other than
415 * '[', ']', '{', '}', '#' and the single quote will be output directly.
416 * Despite this, it is recommended to use single quotes around all characters
417 * that you want to output directly to ensure that future changes do not break
418 * your application.
419 *
420 * <h3 id="resolving">Resolving</h3>
421 * Parsing is implemented as a two-phase operation.
422 * First, the text is parsed using the layout defined by the formatter, producing
423 * a {@code Map} of field to value, a {@code ZoneId} and a {@code Chronology}.
424 * Second, the parsed data is <em>resolved</em>, by validating, combining and
425 * simplifying the various fields into more useful ones.
426 * <p>
427 * Five parsing methods are supplied by this class.
428 * Four of these perform both the parse and resolve phases.
429 * The fifth method, {@link #parseUnresolved(CharSequence, ParsePosition)},
430 * only performs the first phase, leaving the result unresolved.
431 * As such, it is essentially a low-level operation.
432 * <p>
433 * The resolve phase is controlled by two parameters, set on this class.
434 * <p>
435 * The {@link ResolverStyle} is an enum that offers three different approaches,
436 * strict, smart and lenient. The smart option is the default.
437 * It can be set using {@link #withResolverStyle(ResolverStyle)}.
438 * <p>
439 * The {@link #withResolverFields(TemporalField...)} parameter allows the
440 * set of fields that will be resolved to be filtered before resolving starts.
441 * For example, if the formatter has parsed a year, month, day-of-month
442 * and day-of-year, then there are two approaches to resolve a date:

```



```

443 * (year + month + day-of-month) and (year + day-of-year).
444 * The resolver fields allows one of the two approaches to be selected.
445 * If no resolver fields are set then both approaches must result in the same date.
446 * <p>
447 * Resolving separate fields to form a complete date and time is a complex
448 * process with behaviour distributed across a number of classes.
449 * It follows these steps:
450 * <ol>
451 * <li>The chronology is determined.
452 * The chronology of the result is either the chronology that was parsed,
453 * or if no chronology was parsed, it is the chronology set on this class,
454 * or if that is null, it is {@code IsoChronology}.
455 * <li>The {@code ChronoField} date fields are resolved.
456 * This is achieved using {@link Chronology#resolveDate(Map, ResolverStyle)}.
457 * Documentation about field resolution is located in the implementation
458 * of {@code Chronology}.
459 * <li>The {@code ChronoField} time fields are resolved.
460 * This is documented on {@link ChronoField} and is the same for all chronologies.
461 * <li>Any fields that are not {@code ChronoField} are processed.
462 * This is achieved using {@link TemporalField#resolve(Map, TemporalAccessor, ResolverStyle)}.
463 * Documentation about field resolution is located in the implementation
464 * of {@code TemporalField}.
465 * <li>The {@code ChronoField} date and time fields are re-resolved.
466 * This allows fields in step four to produce {@code ChronoField} values
467 * and have them be processed into dates and times.
468 * <li>A {@code LocalDateTime} is formed if there is at least an hour-of-day available.
469 * This involves providing default values for minute, second and fraction of second.
470 * <li>Any remaining unresolved fields are cross-checked against any
471 * date and/or time that was resolved. Thus, an earlier stage would resolve
472 * (year + month + day-of-month) to a date, and this stage would check that
473 * day-of-week was valid for the date.
474 * <li>If an {@link PlainText#parsedExcessDays()} excess number of days}
475 * was parsed then it is added to the date if a date is available.
476 * </ol>
477 *
478 * @implSpec
479 * This class is immutable and thread-safe.
480 *
481 * @since 1.8
482 */
483 public final class DateTimeFormatter {
484
485     /**
486      * The printer and/or parser to use, not null.
487      */
488     private final CompositePrinterParser printerParser;
489     /**
490      * The locale to use for formatting, not null.
491      */
492     private final Locale locale;
493     /**
494      * The symbols to use for formatting, not null.
495      */

```



```

496 private final DecimalStyle decimalStyle;
497 /**
498  * The resolver style to use, not null.
499  */
500 private final ResolverStyle resolverStyle;
501 /**
502  * The fields to use in resolving, null for all fields.
503  */
504 private final Set<TemporalField> resolverFields;
505 /**
506  * The chronology to use for formatting, null for no override.
507  */
508 private final Chronology chrono;
509 /**
510  * The zone to use for formatting, null for no override.
511  */
512 private final ZoneId zone;
513
514 //-----
515 /**
516  * Creates a formatter using the specified pattern.
517  * <p>
518  * This method will create a formatter based on a simple
519  * <a href="#patterns">pattern of letters and symbols</a>
520  * as described in the class documentation.
521  * For example, {@code d MMM uuuu} will format 2011-12-03 as '3 Dec 2011'.
522  * <p>
523  * The formatter will use the {@link Locale#getDefault(Locale.Category) default FORMAT locale}.
524  * This can be changed using {@link DateTimeFormatter#withLocale(Locale)} on the returned
525  * formatter
526  * Alternatively use the {@link #ofPattern(String, Locale)} variant of this method.
527  * <p>
528  * The returned formatter has no override chronology or zone.
529  * It uses {@link ResolverStyle#SMART SMART} resolver style.
530  *
531  * @param pattern the pattern to use, not null
532  * @return the formatter based on the pattern, not null
533  * @throws IllegalArgumentException if the pattern is invalid
534  * @see DateTimeFormatterBuilder#appendPattern(String)
535  */
536 public static DateTimeFormatter ofPattern(String pattern) {
537     return new DateTimeFormatterBuilder().appendPattern(pattern).toFormatter();
538 }
539 /**
540  * Creates a formatter using the specified pattern and locale.
541  * <p>
542  * This method will create a formatter based on a simple
543  * <a href="#patterns">pattern of letters and symbols</a>
544  * as described in the class documentation.
545  * For example, {@code d MMM uuuu} will format 2011-12-03 as '3 Dec 2011'.
546  * <p>
547  * The formatter will use the specified locale.

```

```

548     * This can be changed using {@link DateTimeFormatter#withLocale(Locale)} on the returned
        formatter
549     * <p>
550     * The returned formatter has no override chronology or zone.
551     * It uses {@link ResolverStyle#SMART SMART} resolver style.
552     *
553     * @param pattern the pattern to use, not null
554     * @param locale the locale to use, not null
555     * @return the formatter based on the pattern, not null
556     * @throws IllegalArgumentException if the pattern is invalid
557     * @see DateTimeFormatterBuilder#appendPattern(String)
558     */
559     public static DateTimeFormatter ofPattern(String pattern, Locale locale) {
560         return new DateTimeFormatterBuilder().appendPattern(pattern).toFormatter(locale);
561     }
562
563     //-----
564     /**
565     * Returns a locale specific date format for the ISO chronology.
566     * <p>
567     * This returns a formatter that will format or parse a date.
568     * The exact format pattern used varies by locale.
569     * <p>
570     * The locale is determined from the formatter. The formatter returned directly by
571     * this method will use the {@link Locale#getDefault(Locale.Category) default FORMAT locale}.
572     * The locale can be controlled using {@link DateTimeFormatter#withLocale(Locale) withLocale(
573         Locale)}
574     * on the result of this method.
575     * <p>
576     * Note that the localized pattern is looked up lazily.
577     * This {@code DateTimeFormatter} holds the style required and the locale,
578     * looking up the pattern required on demand.
579     * <p>
580     * The returned formatter has a chronology of ISO set to ensure dates in
581     * other calendar systems are correctly converted.
582     * It has no override zone and uses the {@link ResolverStyle#SMART SMART} resolver style.
583     *
584     * @param dateStyle the formatter style to obtain, not null
585     * @return the date formatter, not null
586     */
587     public static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle) {
588         Objects.requireNonNull(dateStyle, "dateStyle");
589         return new DateTimeFormatterBuilder().appendLocalized(dateStyle, null)
590             .toFormatter(ResolverStyle.SMART, IsoChronology.INSTANCE);
591     }
592
593     /**
594     * Returns a locale specific time format for the ISO chronology.
595     * <p>
596     * This returns a formatter that will format or parse a time.
597     * The exact format pattern used varies by locale.
598     * <p>
599     * The locale is determined from the formatter. The formatter returned directly by

```

```

599     * this method will use the {@link Locale#getDefault(Locale.Category) default FORMAT locale}.
600     * The locale can be controlled using {@link DateTimeFormatter#withLocale(Locale) withLocale(
        Locale)}}
601     * on the result of this method.
602     * <p>
603     * Note that the localized pattern is looked up lazily.
604     * This {@code DateTimeFormatter} holds the style required and the locale,
605     * looking up the pattern required on demand.
606     * <p>
607     * The returned formatter has a chronology of ISO set to ensure dates in
608     * other calendar systems are correctly converted.
609     * It has no override zone and uses the {@link ResolverStyle#SMART SMART} resolver style.
610     *
611     * @param timeStyle the formatter style to obtain, not null
612     * @return the time formatter, not null
613     */
614     public static DateTimeFormatter ofLocalizedTime(FormatStyle timeStyle) {
615         Objects.requireNonNull(timeStyle, "timeStyle");
616         return new DateTimeFormatterBuilder().appendLocalized(null, timeStyle)
617             .toFormatter(ResolverStyle.SMART, IsoChronology.INSTANCE);
618     }
619
620     /**
621     * Returns a locale specific date-time formatter for the ISO chronology.
622     * <p>
623     * This returns a formatter that will format or parse a date-time.
624     * The exact format pattern used varies by locale.
625     * <p>
626     * The locale is determined from the formatter. The formatter returned directly by
627     * this method will use the {@link Locale#getDefault(Locale.Category) default FORMAT locale}.
628     * The locale can be controlled using {@link DateTimeFormatter#withLocale(Locale) withLocale(
        Locale)}}
629     * on the result of this method.
630     * <p>
631     * Note that the localized pattern is looked up lazily.
632     * This {@code DateTimeFormatter} holds the style required and the locale,
633     * looking up the pattern required on demand.
634     * <p>
635     * The returned formatter has a chronology of ISO set to ensure dates in
636     * other calendar systems are correctly converted.
637     * It has no override zone and uses the {@link ResolverStyle#SMART SMART} resolver style.
638     *
639     * @param dateTimeStyle the formatter style to obtain, not null
640     * @return the date-time formatter, not null
641     */
642     public static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTimeStyle) {
643         Objects.requireNonNull(dateTimeStyle, "dateTimeStyle");
644         return new DateTimeFormatterBuilder().appendLocalized(dateTimeStyle, dateTimeStyle)
645             .toFormatter(ResolverStyle.SMART, IsoChronology.INSTANCE);
646     }
647
648     /**
649     * Returns a locale specific date and time format for the ISO chronology.

```

```

650 * <p>
651 * This returns a formatter that will format or parse a date-time.
652 * The exact format pattern used varies by locale.
653 * <p>
654 * The locale is determined from the formatter. The formatter returned directly by
655 * this method will use the {@link Locale#getDefault() default FORMAT locale}.
656 * The locale can be controlled using {@link DateTimeFormatter#withLocale(Locale) withLocale(
        Locale)}
657 * on the result of this method.
658 * <p>
659 * Note that the localized pattern is looked up lazily.
660 * This {@code DateTimeFormatter} holds the style required and the locale,
661 * looking up the pattern required on demand.
662 * <p>
663 * The returned formatter has a chronology of ISO set to ensure dates in
664 * other calendar systems are correctly converted.
665 * It has no override zone and uses the {@link ResolverStyle#SMART SMART} resolver style.
666 *
667 * @param dateStyle the date formatter style to obtain, not null
668 * @param timeStyle the time formatter style to obtain, not null
669 * @return the date, time or date-time formatter, not null
670 */
671 public static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateStyle, FormatStyle
        timeStyle) {
672     Objects.requireNonNull(dateStyle, "dateStyle");
673     Objects.requireNonNull(timeStyle, "timeStyle");
674     return new DateTimeFormatterBuilder().appendLocalized(dateStyle, timeStyle)
675         .toFormatter(ResolverStyle.SMART, IsoChronology.INSTANCE);
676 }
677
678 //-----
679 /**
680 * The ISO date formatter that formats or parses a date without an
681 * offset, such as '2011-12-03'.
682 * <p>
683 * This returns an immutable formatter capable of formatting and parsing
684 * the ISO-8601 extended local date format.
685 * The format consists of:
686 * <ul>
687 * <li>Four digits or more for the {@link ChronoField#YEAR year}.
688 * Years in the range 0000 to 9999 will be pre-padded by zero to ensure four digits.
689 * Years outside that range will have a prefixed positive or negative symbol.
690 * <li>A dash
691 * <li>Two digits for the {@link ChronoField#MONTH_OF_YEAR month-of-year}.
692 * This is pre-padded by zero to ensure two digits.
693 * <li>A dash
694 * <li>Two digits for the {@link ChronoField#DAY_OF_MONTH day-of-month}.
695 * This is pre-padded by zero to ensure two digits.
696 * </ul>
697 * <p>
698 * The returned formatter has a chronology of ISO set to ensure dates in
699 * other calendar systems are correctly converted.
700 * It has no override zone and uses the {@link ResolverStyle#STRICT STRICT} resolver style.

```

```

701     */
702     public static final DateTimeFormatter ISO_LOCAL_DATE;
703     static {
704         ISO_LOCAL_DATE = new DateTimeFormatterBuilder()
705             .appendValue(YEAR, 4, 10, SignStyle.EXCEEDS_PAD)
706             .appendLiteral('-')
707             .appendValue(MONTH_OF_YEAR, 2)
708             .appendLiteral('-')
709             .appendValue(DAY_OF_MONTH, 2)
710             .toFormatter(ResolverStyle.STRICT, IsoChronology.INSTANCE);
711     }
712
713     //-----
714     /**
715      * The ISO date formatter that formats or parses a date with an
716      * offset, such as '2011-12-03+01:00'.
717      * <p>
718      * This returns an immutable formatter capable of formatting and parsing
719      * the ISO-8601 extended offset date format.
720      * The format consists of:
721      * <ul>
722      * <li>The {@link #ISO_LOCAL_DATE}
723      * <li>The {@link ZoneOffset#getId() offset ID}. If the offset has seconds then
724      * they will be handled even though this is not part of the ISO-8601 standard.
725      * Parsing is case insensitive.
726      * </ul>
727      * <p>
728      * The returned formatter has a chronology of ISO set to ensure dates in
729      * other calendar systems are correctly converted.
730      * It has no override zone and uses the {@link ResolverStyle#STRICT STRICT} resolver style.
731      */
732     public static final DateTimeFormatter ISO_OFFSET_DATE;
733     static {
734         ISO_OFFSET_DATE = new DateTimeFormatterBuilder()
735             .parseCaseInsensitive()
736             .append(ISO_LOCAL_DATE)
737             .appendOffsetId()
738             .toFormatter(ResolverStyle.STRICT, IsoChronology.INSTANCE);
739     }
740
741     //-----
742     /**
743      * The ISO date formatter that formats or parses a date with the
744      * offset if available, such as '2011-12-03' or '2011-12-03+01:00'.
745      * <p>
746      * This returns an immutable formatter capable of formatting and parsing
747      * the ISO-8601 extended date format.
748      * The format consists of:
749      * <ul>
750      * <li>The {@link #ISO_LOCAL_DATE}
751      * <li>If the offset is not available then the format is complete.
752      * <li>The {@link ZoneOffset#getId() offset ID}. If the offset has seconds then
753      * they will be handled even though this is not part of the ISO-8601 standard.

```

```

754     * Parsing is case insensitive.
755     * </ul>
756     * <p>
757     * As this formatter has an optional element, it may be necessary to parse using
758     * {@link DateTimeFormatter#parseBest}.
759     * <p>
760     * The returned formatter has a chronology of ISO set to ensure dates in
761     * other calendar systems are correctly converted.
762     * It has no override zone and uses the {@link ResolverStyle#STRICT STRICT} resolver style.
763     */
764     public static final DateTimeFormatter ISO_DATE;
765     static {
766         ISO_DATE = new DateTimeFormatterBuilder()
767             .parseCaseInsensitive()
768             .append(ISO_LOCAL_DATE)
769             .optionalStart()
770             .appendOffsetId()
771             .toFormatter(ResolverStyle.STRICT, IsoChronology.INSTANCE);
772     }
773
774     //-----
775     /**
776     * The ISO time formatter that formats or parses a time without an
777     * offset, such as '10:15' or '10:15:30'.
778     * <p>
779     * This returns an immutable formatter capable of formatting and parsing
780     * the ISO-8601 extended local time format.
781     * The format consists of:
782     * <ul>
783     * <li>Two digits for the {@link ChronoField#HOUR_OF_DAY hour-of-day}.
784     * This is pre-padded by zero to ensure two digits.
785     * <li>A colon
786     * <li>Two digits for the {@link ChronoField#MINUTE_OF_HOUR minute-of-hour}.
787     * This is pre-padded by zero to ensure two digits.
788     * <li>If the second-of-minute is not available then the format is complete.
789     * <li>A colon
790     * <li>Two digits for the {@link ChronoField#SECOND_OF_MINUTE second-of-minute}.
791     * This is pre-padded by zero to ensure two digits.
792     * <li>If the nano-of-second is zero or not available then the format is complete.
793     * <li>A decimal point
794     * <li>One to nine digits for the {@link ChronoField#NANO_OF_SECOND nano-of-second}.
795     * As many digits will be output as required.
796     * </ul>
797     * <p>
798     * The returned formatter has no override chronology or zone.
799     * It uses the {@link ResolverStyle#STRICT STRICT} resolver style.
800     */
801     public static final DateTimeFormatter ISO_LOCAL_TIME;
802     static {
803         ISO_LOCAL_TIME = new DateTimeFormatterBuilder()
804             .appendValue(HOUR_OF_DAY, 2)
805             .appendLiteral(':')
806             .appendValue(MINUTE_OF_HOUR, 2)

```

```

807         .optionalStart()
808         .appendLiteral(':')
809         .appendValue(SECOND_OF_MINUTE, 2)
810         .optionalStart()
811         .appendFraction(NANO_OF_SECOND, 0, 9, true)
812         .toFormatter(ResolverStyle.STRICT, null);
813     }
814
815     //-----
816     /**
817      * The ISO time formatter that formats or parses a time with an
818      * offset, such as '10:15+01:00' or '10:15:30+01:00'.
819      * <p>
820      * This returns an immutable formatter capable of formatting and parsing
821      * the ISO-8601 extended offset time format.
822      * The format consists of:
823      * <ul>
824      * <li>The {@link #ISO_LOCAL_TIME}
825      * <li>The {@link ZoneOffset#getId() offset ID}. If the offset has seconds then
826      * they will be handled even though this is not part of the ISO-8601 standard.
827      * Parsing is case insensitive.
828      * </ul>
829      * <p>
830      * The returned formatter has no override chronology or zone.
831      * It uses the {@link ResolverStyle#STRICT STRICT} resolver style.
832      */
833     public static final DateTimeFormatter ISO_OFFSET_TIME;
834     static {
835         ISO_OFFSET_TIME = new DateTimeFormatterBuilder()
836             .parseCaseInsensitive()
837             .append(ISO_LOCAL_TIME)
838             .appendOffsetId()
839             .toFormatter(ResolverStyle.STRICT, null);
840     }
841
842     //-----
843     /**
844      * The ISO time formatter that formats or parses a time, with the
845      * offset if available, such as '10:15', '10:15:30' or '10:15:30+01:00'.
846      * <p>
847      * This returns an immutable formatter capable of formatting and parsing
848      * the ISO-8601 extended offset time format.
849      * The format consists of:
850      * <ul>
851      * <li>The {@link #ISO_LOCAL_TIME}
852      * <li>If the offset is not available then the format is complete.
853      * <li>The {@link ZoneOffset#getId() offset ID}. If the offset has seconds then
854      * they will be handled even though this is not part of the ISO-8601 standard.
855      * Parsing is case insensitive.
856      * </ul>
857      * <p>
858      * As this formatter has an optional element, it may be necessary to parse using
859      * {@link DateTimeFormatter#parseBest}.

```

```

860     * <p>
861     * The returned formatter has no override chronology or zone.
862     * It uses the {@link ResolverStyle#STRICT STRICT} resolver style.
863     */
864     public static final DateTimeFormatter ISO_TIME;
865     static {
866         ISO_TIME = new DateTimeFormatterBuilder()
867             .parseCaseInsensitive()
868             .append(ISO_LOCAL_TIME)
869             .optionalStart()
870             .appendOffsetId()
871             .toFormatter(ResolverStyle.STRICT, null);
872     }
873
874     //-----
875     /**
876     * The ISO date-time formatter that formats or parses a date-time without
877     * an offset, such as '2011-12-03T10:15:30'.
878     * <p>
879     * This returns an immutable formatter capable of formatting and parsing
880     * the ISO-8601 extended offset date-time format.
881     * The format consists of:
882     * <ul>
883     * <li>The {@link #ISO_LOCAL_DATE}
884     * <li>The letter 'T'. Parsing is case insensitive.
885     * <li>The {@link #ISO_LOCAL_TIME}
886     * </ul>
887     * <p>
888     * The returned formatter has a chronology of ISO set to ensure dates in
889     * other calendar systems are correctly converted.
890     * It has no override zone and uses the {@link ResolverStyle#STRICT STRICT} resolver style.
891     */
892     public static final DateTimeFormatter ISO_LOCAL_DATE_TIME;
893     static {
894         ISO_LOCAL_DATE_TIME = new DateTimeFormatterBuilder()
895             .parseCaseInsensitive()
896             .append(ISO_LOCAL_DATE)
897             .appendLiteral('T')
898             .append(ISO_LOCAL_TIME)
899             .toFormatter(ResolverStyle.STRICT, IsoChronology.INSTANCE);
900     }
901
902     //-----
903     /**
904     * The ISO date-time formatter that formats or parses a date-time with an
905     * offset, such as '2011-12-03T10:15:30+01:00'.
906     * <p>
907     * This returns an immutable formatter capable of formatting and parsing
908     * the ISO-8601 extended offset date-time format.
909     * The format consists of:
910     * <ul>
911     * <li>The {@link #ISO_LOCAL_DATE_TIME}
912     * <li>The {@link ZoneOffset#getId() offset ID}. If the offset has seconds then

```



```

913     * they will be handled even though this is not part of the ISO-8601 standard.
914     * Parsing is case insensitive.
915     * </ul>
916     * <p>
917     * The returned formatter has a chronology of ISO set to ensure dates in
918     * other calendar systems are correctly converted.
919     * It has no override zone and uses the {@link ResolverStyle#STRICT STRICT} resolver style.
920     */
921     public static final DateTimeFormatter ISO_OFFSET_DATE_TIME;
922     static {
923         ISO_OFFSET_DATE_TIME = new DateTimeFormatterBuilder()
924             .parseCaseInsensitive()
925             .append(ISO_LOCAL_DATE_TIME)
926             .appendOffsetId()
927             .toFormatter(ResolverStyle.STRICT, IsoChronology.INSTANCE);
928     }
929
930     //-----
931     /**
932     * The ISO-like date-time formatter that formats or parses a date-time with
933     * offset and zone, such as '2011-12-03T10:15:30+01:00[Europe/Paris]'.
934     * <p>
935     * This returns an immutable formatter capable of formatting and parsing
936     * a format that extends the ISO-8601 extended offset date-time format
937     * to add the time-zone.
938     * The section in square brackets is not part of the ISO-8601 standard.
939     * The format consists of:
940     * <ul>
941     * <li>The {@link #ISO_OFFSET_DATE_TIME}
942     * <li>If the zone ID is not available or is a {@code ZoneOffset} then the format is complete.
943     * <li>An open square bracket '['.
944     * <li>The {@link ZoneId#getId() zone ID}. This is not part of the ISO-8601 standard.
945     * <li>Parsing is case sensitive.
946     * <li>A close square bracket ']'.
947     * </ul>
948     * <p>
949     * The returned formatter has a chronology of ISO set to ensure dates in
950     * other calendar systems are correctly converted.
951     * It has no override zone and uses the {@link ResolverStyle#STRICT STRICT} resolver style.
952     */
953     public static final DateTimeFormatter ISO_ZONED_DATE_TIME;
954     static {
955         ISO_ZONED_DATE_TIME = new DateTimeFormatterBuilder()
956             .append(ISO_OFFSET_DATE_TIME)
957             .optionalStart()
958             .appendLiteral('[')
959             .parseCaseSensitive()
960             .appendZoneRegionId()
961             .appendLiteral(']')
962             .toFormatter(ResolverStyle.STRICT, IsoChronology.INSTANCE);
963     }
964
965     //-----

```

```

966 /**
967  * The ISO-like date-time formatter that formats or parses a date-time with
968  * the offset and zone if available, such as '2011-12-03T10:15:30',
969  * '2011-12-03T10:15:30+01:00' or '2011-12-03T10:15:30+01:00[Europe/Paris]'.
970  * <p>
971  * This returns an immutable formatter capable of formatting and parsing
972  * the ISO-8601 extended local or offset date-time format, as well as the
973  * extended non-ISO form specifying the time-zone.
974  * The format consists of:
975  * <ul>
976  * <li>The {@link #ISO_LOCAL_DATE_TIME}
977  * <li>If the offset is not available to format or parse then the format is complete.
978  * <li>The {@link ZoneOffset#getId() offset ID}. If the offset has seconds then
979  * they will be handled even though this is not part of the ISO-8601 standard.
980  * <li>If the zone ID is not available or is a {@code ZoneOffset} then the format is complete.
981  * <li>An open square bracket '['.
982  * <li>The {@link ZoneId#getId() zone ID}. This is not part of the ISO-8601 standard.
983  * Parsing is case sensitive.
984  * <li>A close square bracket ']'.
985  * </ul>
986  * <p>
987  * As this formatter has an optional element, it may be necessary to parse using
988  * {@link DateTimeFormatter#parseBest}.
989  * <p>
990  * The returned formatter has a chronology of ISO set to ensure dates in
991  * other calendar systems are correctly converted.
992  * It has no override zone and uses the {@link ResolverStyle#STRICT STRICT} resolver style.
993  */
994 public static final DateTimeFormatter ISO_DATE_TIME;
995 static {
996     ISO_DATE_TIME = new DateTimeFormatterBuilder()
997         .append(ISO_LOCAL_DATE_TIME)
998         .optionalStart()
999         .appendOffsetId()
1000         .optionalStart()
1001         .appendLiteral('[')
1002         .parseCaseSensitive()
1003         .appendZoneRegionId()
1004         .appendLiteral(']')
1005         .toFormatter(ResolverStyle.STRICT, IsoChronology.INSTANCE);
1006 }
1007
1008 //-----
1009 /**
1010  * The ISO date formatter that formats or parses the ordinal date
1011  * without an offset, such as '2012-337'.
1012  * <p>
1013  * This returns an immutable formatter capable of formatting and parsing
1014  * the ISO-8601 extended ordinal date format.
1015  * The format consists of:
1016  * <ul>
1017  * <li>Four digits or more for the {@link ChronoField#YEAR year}.
1018  * Years in the range 0000 to 9999 will be pre-padded by zero to ensure four digits.

```

```

1019 * Years outside that range will have a prefixed positive or negative symbol.
1020 * <li>A dash
1021 * <li>Three digits for the {@link ChronoField#DAY_OF_YEAR day-of-year}.
1022 * This is pre-padded by zero to ensure three digits.
1023 * <li>If the offset is not available to format or parse then the format is complete.
1024 * <li>The {@link ZoneOffset#getId() offset ID}. If the offset has seconds then
1025 * they will be handled even though this is not part of the ISO-8601 standard.
1026 * Parsing is case insensitive.
1027 * </ul>
1028 * <p>
1029 * As this formatter has an optional element, it may be necessary to parse using
1030 * {@link DateTimeFormatter#parseBest}.
1031 * <p>
1032 * The returned formatter has a chronology of ISO set to ensure dates in
1033 * other calendar systems are correctly converted.
1034 * It has no override zone and uses the {@link ResolverStyle#STRICT STRICT} resolver style.
1035 */
1036 public static final DateTimeFormatter ISO_ORDINAL_DATE;
1037 static {
1038     ISO_ORDINAL_DATE = new DateTimeFormatterBuilder()
1039         .parseCaseInsensitive()
1040         .appendValue(YEAR, 4, 10, SignStyle.EXCEEDS_PAD)
1041         .appendLiteral('-')
1042         .appendValue(DAY_OF_YEAR, 3)
1043         .optionalStart()
1044         .appendOffsetId()
1045         .toFormatter(ResolverStyle.STRICT, IsoChronology.INSTANCE);
1046 }
1047
1048 //-----
1049 /**
1050 * The ISO date formatter that formats or parses the week-based date
1051 * without an offset, such as '2012-W48-6'.
1052 * <p>
1053 * This returns an immutable formatter capable of formatting and parsing
1054 * the ISO-8601 extended week-based date format.
1055 * The format consists of:
1056 * <ul>
1057 * <li>Four digits or more for the {@link IsoFields#WEEK_BASED_YEAR week-based-year}.
1058 * Years in the range 0000 to 9999 will be pre-padded by zero to ensure four digits.
1059 * Years outside that range will have a prefixed positive or negative symbol.
1060 * <li>A dash
1061 * <li>The letter 'W'. Parsing is case insensitive.
1062 * <li>Two digits for the {@link IsoFields#WEEK_OF_WEEK_BASED_YEAR week-of-week-based-year}.
1063 * This is pre-padded by zero to ensure three digits.
1064 * <li>A dash
1065 * <li>One digit for the {@link ChronoField#DAY_OF_WEEK day-of-week}.
1066 * The value run from Monday (1) to Sunday (7).
1067 * <li>If the offset is not available to format or parse then the format is complete.
1068 * <li>The {@link ZoneOffset#getId() offset ID}. If the offset has seconds then
1069 * they will be handled even though this is not part of the ISO-8601 standard.
1070 * Parsing is case insensitive.
1071 * </ul>

```

```

1072 * <p>
1073 * As this formatter has an optional element, it may be necessary to parse using
1074 * {@link DateTimeFormatter#parseBest}.
1075 * <p>
1076 * The returned formatter has a chronology of ISO set to ensure dates in
1077 * other calendar systems are correctly converted.
1078 * It has no override zone and uses the {@link ResolverStyle#STRICT STRICT} resolver style.
1079 */
1080 public static final DateTimeFormatter ISO_WEEK_DATE;
1081 static {
1082     ISO_WEEK_DATE = new DateTimeFormatterBuilder()
1083         .parseCaseInsensitive()
1084         .appendValue(IsoFields.WEEK_BASED_YEAR, 4, 10, SignStyle.EXCEEDS_PAD)
1085         .appendLiteral("-W")
1086         .appendValue(IsoFields.WEEK_OF_WEEK_BASED_YEAR, 2)
1087         .appendLiteral('-')
1088         .appendValue(DAY_OF_WEEK, 1)
1089         .optionalStart()
1090         .appendOffsetId()
1091         .toFormatter(ResolverStyle.STRICT, IsoChronology.INSTANCE);
1092 }
1093
1094 //-----
1095 /**
1096 * The ISO instant formatter that formats or parses an instant in UTC,
1097 * such as '2011-12-03T10:15:30Z'.
1098 * <p>
1099 * This returns an immutable formatter capable of formatting and parsing
1100 * the ISO-8601 instant format.
1101 * When formatting, the second-of-minute is always output.
1102 * The nano-of-second outputs zero, three, six or nine digits as necessary.
1103 * When parsing, time to at least the seconds field is required.
1104 * Fractional seconds from zero to nine are parsed.
1105 * The localized decimal style is not used.
1106 * <p>
1107 * This is a special case formatter intended to allow a human readable form
1108 * of an {@link java.time.Instant}. The {@code Instant} class is designed to
1109 * only represent a point in time and internally stores a value in nanoseconds
1110 * from a fixed epoch of 1970-01-01Z. As such, an {@code Instant} cannot be
1111 * formatted as a date or time without providing some form of time-zone.
1112 * This formatter allows the {@code Instant} to be formatted, by providing
1113 * a suitable conversion using {@code ZoneOffset.UTC}.
1114 * <p>
1115 * The format consists of:
1116 * <ul>
1117 * <li>The {@link #ISO_OFFSET_DATE_TIME} where the instant is converted from
1118 *   {@link ChronoField#INSTANT_SECONDS} and {@link ChronoField#NANO_OF_SECOND}
1119 *   using the {@code UTC} offset. Parsing is case insensitive.
1120 * </ul>
1121 * <p>
1122 * The returned formatter has no override chronology or zone.
1123 * It uses the {@link ResolverStyle#STRICT STRICT} resolver style.
1124 */

```

```

1125 public static final DateTimeFormatter ISO_INSTANT;
1126 static {
1127     ISO_INSTANT = new DateTimeFormatterBuilder()
1128         .parseCaseInsensitive()
1129         .appendInstant()
1130         .toFormatter(ResolverStyle.STRICT, null);
1131 }
1132
1133 //-----
1134 /**
1135  * The ISO date formatter that formats or parses a date without an
1136  * offset, such as '20111203'.
1137  * <p>
1138  * This returns an immutable formatter capable of formatting and parsing
1139  * the ISO-8601 basic local date format.
1140  * The format consists of:
1141  * <ul>
1142  * <li>Four digits for the {@link ChronoField#YEAR year}.
1143  * Only years in the range 0000 to 9999 are supported.
1144  * <li>Two digits for the {@link ChronoField#MONTH_OF_YEAR month-of-year}.
1145  * This is pre-padded by zero to ensure two digits.
1146  * <li>Two digits for the {@link ChronoField#DAY_OF_MONTH day-of-month}.
1147  * This is pre-padded by zero to ensure two digits.
1148  * <li>If the offset is not available to format or parse then the format is complete.
1149  * <li>The {@link ZoneOffset#getId() offset ID} without colons. If the offset has
1150  * seconds then they will be handled even though this is not part of the ISO-8601 standard.
1151  * Parsing is case insensitive.
1152  * </ul>
1153  * <p>
1154  * As this formatter has an optional element, it may be necessary to parse using
1155  * {@link DateTimeFormatter#parseBest}.
1156  * <p>
1157  * The returned formatter has a chronology of ISO set to ensure dates in
1158  * other calendar systems are correctly converted.
1159  * It has no override zone and uses the {@link ResolverStyle#STRICT STRICT} resolver style.
1160  */
1161 public static final DateTimeFormatter BASIC_ISO_DATE;
1162 static {
1163     BASIC_ISO_DATE = new DateTimeFormatterBuilder()
1164         .parseCaseInsensitive()
1165         .appendValue(YEAR, 4)
1166         .appendValue(MONTH_OF_YEAR, 2)
1167         .appendValue(DAY_OF_MONTH, 2)
1168         .optionalStart()
1169         .appendOffset("+HHMMss", "Z")
1170         .toFormatter(ResolverStyle.STRICT, IsoChronology.INSTANCE);
1171 }
1172
1173 //-----
1174 /**
1175  * The RFC-1123 date-time formatter, such as 'Tue, 3 Jun 2008 11:05:30 GMT'.
1176  * <p>
1177  * This returns an immutable formatter capable of formatting and parsing

```

```

1178 * most of the RFC-1123 format.
1179 * RFC-1123 updates RFC-822 changing the year from two digits to four.
1180 * This implementation requires a four digit year.
1181 * This implementation also does not handle North American or military zone
1182 * names, only 'GMT' and offset amounts.
1183 * <p>
1184 * The format consists of:
1185 * <ul>
1186 * <li>If the day-of-week is not available to format or parse then jump to day-of-month.
1187 * <li>Three letter {@link ChronoField#DAY_OF_WEEK day-of-week} in English.
1188 * <li>A comma
1189 * <li>A space
1190 * <li>One or two digits for the {@link ChronoField#DAY_OF_MONTH day-of-month}.
1191 * <li>A space
1192 * <li>Three letter {@link ChronoField#MONTH_OF_YEAR month-of-year} in English.
1193 * <li>A space
1194 * <li>Four digits for the {@link ChronoField#YEAR year}.
1195 * Only years in the range 0000 to 9999 are supported.
1196 * <li>A space
1197 * <li>Two digits for the {@link ChronoField#HOUR_OF_DAY hour-of-day}.
1198 * This is pre-padded by zero to ensure two digits.
1199 * <li>A colon
1200 * <li>Two digits for the {@link ChronoField#MINUTE_OF_HOUR minute-of-hour}.
1201 * This is pre-padded by zero to ensure two digits.
1202 * <li>If the second-of-minute is not available then jump to the next space.
1203 * <li>A colon
1204 * <li>Two digits for the {@link ChronoField#SECOND_OF_MINUTE second-of-minute}.
1205 * This is pre-padded by zero to ensure two digits.
1206 * <li>A space
1207 * <li>The {@link ZoneOffset#getId() offset ID} without colons or seconds.
1208 * An offset of zero uses "GMT". North American zone names and military zone names are not
    handled.
1209 * </ul>
1210 * <p>
1211 * Parsing is case insensitive.
1212 * <p>
1213 * The returned formatter has a chronology of ISO set to ensure dates in
1214 * other calendar systems are correctly converted.
1215 * It has no override zone and uses the {@link ResolverStyle#SMART SMART} resolver style.
1216 */
1217 public static final DateTimeFormatter RFC_1123_DATE_TIME;
1218 static {
1219     // manually code maps to ensure correct data always used
1220     // (locale data can be changed by application code)
1221     Map<Long, String> dow = new HashMap<>();
1222     dow.put(1L, "Mon");
1223     dow.put(2L, "Tue");
1224     dow.put(3L, "Wed");
1225     dow.put(4L, "Thu");
1226     dow.put(5L, "Fri");
1227     dow.put(6L, "Sat");
1228     dow.put(7L, "Sun");
1229     Map<Long, String> moy = new HashMap<>();

```

```

1230     moy.put(1L, "Jan");
1231     moy.put(2L, "Feb");
1232     moy.put(3L, "Mar");
1233     moy.put(4L, "Apr");
1234     moy.put(5L, "May");
1235     moy.put(6L, "Jun");
1236     moy.put(7L, "Jul");
1237     moy.put(8L, "Aug");
1238     moy.put(9L, "Sep");
1239     moy.put(10L, "Oct");
1240     moy.put(11L, "Nov");
1241     moy.put(12L, "Dec");
1242     RFC_1123_DATE_TIME = new DateTimeFormatterBuilder()
1243         .parseCaseInsensitive()
1244         .parseLenient()
1245         .optionalStart()
1246         .appendText(DAY_OF_WEEK, dow)
1247         .appendLiteral(", ")
1248         .optionalEnd()
1249         .appendValue(DAY_OF_MONTH, 1, 2, SignStyle.NOT_NEGATIVE)
1250         .appendLiteral(' ')
1251         .appendText(MONTH_OF_YEAR, moy)
1252         .appendLiteral(' ')
1253         .appendValue(YEAR, 4) // 2 digit year not handled
1254         .appendLiteral(' ')
1255         .appendValue(HOUR_OF_DAY, 2)
1256         .appendLiteral(':')
1257         .appendValue(MINUTE_OF_HOUR, 2)
1258         .optionalStart()
1259         .appendLiteral(':')
1260         .appendValue(SECOND_OF_MINUTE, 2)
1261         .optionalEnd()
1262         .appendLiteral(' ')
1263         .appendOffset("+HHMM", "GMT") // should handle UT/Z/EST/EDT/CST/CDT/MST/MDT/PST/
            MDT
1264         .toFormatter(ResolverStyle.SMART, IsoChronology.INSTANCE);
1265 }
1266
1267 //-----
1268 /**
1269  * A query that provides access to the excess days that were parsed.
1270  * <p>
1271  * This returns a singleton {@linkplain TemporalQuery query} that provides
1272  * access to additional information from the parse. The query always returns
1273  * a non-null period, with a zero period returned instead of null.
1274  * <p>
1275  * There are two situations where this query may return a non-zero period.
1276  * <ul>
1277  * <li>If the {@code ResolverStyle} is {@code LENIENT} and a time is parsed
1278  * without a date, then the complete result of the parse consists of a
1279  * {@code LocalDateTime} and an excess {@code Period} in days.
1280  *
1281  * <li>If the {@code ResolverStyle} is {@code SMART} and a time is parsed

```



```

1282 * without a date where the time is 24:00:00, then the complete result of
1283 * the parse consists of a {@code LocalTime} of 00:00:00 and an excess
1284 * {@code Period} of one day.
1285 * </ul>
1286 * <p>
1287 * In both cases, if a complete {@code ChronoLocalDateTime} or {@code Instant}
1288 * is parsed, then the excess days are added to the date part.
1289 * As a result, this query will return a zero period.
1290 * <p>
1291 * The {@code SMART} behaviour handles the common "end of day" 24:00 value.
1292 * Processing in {@code LENIENT} mode also produces the same result:
1293 * <pre>
1294 * Text to parse      Parsed object      Excess days
1295 * "2012-12-03T00:00"  LocalDateTime.of(2012, 12, 3, 0, 0)  ZERO
1296 * "2012-12-03T24:00"  LocalDateTime.of(2012, 12, 4, 0, 0)  ZERO
1297 * "00:00"            LocalTime.of(0, 0)                ZERO
1298 * "24:00"            LocalTime.of(0, 0)                Period.ofDays(1)
1299 * </pre>
1300 * The query can be used as follows:
1301 * <pre>
1302 * TemporalAccessor parsed = formatter.parse(str);
1303 * LocalTime time = parsed.query(LocalTime::from);
1304 * Period extraDays = parsed.query(DateTimeFormatter.parsedExcessDays());
1305 * </pre>
1306 * @return a query that provides access to the excess days that were parsed
1307 */
1308 public static final TemporalQuery<Period> parsedExcessDays() {
1309     return PARSED_EXCESS_DAYS;
1310 }
1311 private static final TemporalQuery<Period> PARSED_EXCESS_DAYS = t -> {
1312     if (t instanceof Parsed) {
1313         return ((Parsed) t).excessDays;
1314     } else {
1315         return Period.ZERO;
1316     }
1317 };
1318
1319 /**
1320 * A query that provides access to whether a leap-second was parsed.
1321 * <p>
1322 * This returns a singleton {@linkplain TemporalQuery query} that provides
1323 * access to additional information from the parse. The query always returns
1324 * a non-null boolean, true if parsing saw a leap-second, false if not.
1325 * <p>
1326 * Instant parsing handles the special "leap second" time of '23:59:60'.
1327 * Leap seconds occur at '23:59:60' in the UTC time-zone, but at other
1328 * local times in different time-zones. To avoid this potential ambiguity,
1329 * the handling of leap-seconds is limited to
1330 * {@link DateTimeFormatterBuilder#appendInstant()}, as that method
1331 * always parses the instant with the UTC zone offset.
1332 * <p>
1333 * If the time '23:59:60' is received, then a simple conversion is applied,
1334 * replacing the second-of-minute of 60 with 59. This query can be used

```



```

1335     * on the parse result to determine if the leap-second adjustment was made.
1336     * The query will return {@code true} if it did adjust to remove the
1337     * leap-second, and {@code false} if not. Note that applying a leap-second
1338     * smoothing mechanism, such as UTC-SLS, is the responsibility of the
1339     * application, as follows:
1340     * <pre>
1341     * TemporalAccessor parsed = formatter.parse(str);
1342     * Instant instant = parsed.query(Instant::from);
1343     * if (parsed.query(DateTimeFormatter.parsedLeapSecond())) {
1344     *     // validate leap-second is correct and apply correct smoothing
1345     * }
1346     * </pre>
1347     * @return a query that provides access to whether a leap-second was parsed
1348     */
1349     public static final TemporalQuery<Boolean> parsedLeapSecond() {
1350         return PARSED_LEAP_SECOND;
1351     }
1352     private static final TemporalQuery<Boolean> PARSED_LEAP_SECOND = t -> {
1353         if (t instanceof Parsed) {
1354             return ((Parsed) t).leapSecond;
1355         } else {
1356             return Boolean.FALSE;
1357         }
1358     };
1359
1360     //-----
1361     /**
1362     * Constructor.
1363     *
1364     * @param printerParser the printer/parser to use, not null
1365     * @param locale the locale to use, not null
1366     * @param decimalStyle the DecimalStyle to use, not null
1367     * @param resolverStyle the resolver style to use, not null
1368     * @param resolverFields the fields to use during resolving, null for all fields
1369     * @param chrono the chronology to use, null for no override
1370     * @param zone the zone to use, null for no override
1371     */
1372     DateTimeFormatter(CompositePrinterParser printerParser,
1373         Locale locale, DecimalStyle decimalStyle,
1374         ResolverStyle resolverStyle, Set<TemporalField> resolverFields,
1375         Chronology chrono, ZoneId zone) {
1376         this.printerParser = Objects.requireNonNull(printerParser, "printerParser");
1377         this.resolverFields = resolverFields;
1378         this.locale = Objects.requireNonNull(locale, "locale");
1379         this.decimalStyle = Objects.requireNonNull(decimalStyle, "decimalStyle");
1380         this.resolverStyle = Objects.requireNonNull(resolverStyle, "resolverStyle");
1381         this.chrono = chrono;
1382         this.zone = zone;
1383     }
1384
1385     //-----
1386     /**
1387     * Gets the locale to be used during formatting.

```

```
1388 * <p>
1389 * This is used to lookup any part of the formatter needing specific
1390 * localization, such as the text or localized pattern.
1391 *
1392 * @return the locale of this formatter, not null
1393 */
1394 public Locale getLocale() {
1395     return locale;
1396 }
1397
1398 /**
1399 * Returns a copy of this formatter with a new locale.
1400 * <p>
1401 * This is used to lookup any part of the formatter needing specific
1402 * localization, such as the text or localized pattern.
1403 * <p>
1404 * This instance is immutable and unaffected by this method call.
1405 *
1406 * @param locale the new locale, not null
1407 * @return a formatter based on this formatter with the requested locale, not null
1408 */
1409 public DateTimeFormatter withLocale(Locale locale) {
1410     if (this.locale.equals(locale)) {
1411         return this;
1412     }
1413     return new DateTimeFormatter(printerParser, locale, decimalStyle, resolverStyle,
1414         resolverFields, chrono, zone);
1415 }
1416
1417 //-----
1418 /**
1419 * Gets the DecimalStyle to be used during formatting.
1420 *
1421 * @return the locale of this formatter, not null
1422 */
1423 public DecimalStyle getDecimalStyle() {
1424     return decimalStyle;
1425 }
1426
1427 /**
1428 * Returns a copy of this formatter with a new DecimalStyle.
1429 * <p>
1430 * This instance is immutable and unaffected by this method call.
1431 *
1432 * @param decimalStyle the new DecimalStyle, not null
1433 * @return a formatter based on this formatter with the requested DecimalStyle, not null
1434 */
1435 public DateTimeFormatter withDecimalStyle(DecimalStyle decimalStyle) {
1436     if (this.decimalStyle.equals(decimalStyle)) {
1437         return this;
1438     }
1439     return new DateTimeFormatter(printerParser, locale, decimalStyle, resolverStyle,
1440         resolverFields, chrono, zone);
1441 }
```

```

1439     }
1440
1441     //-----
1442     /**
1443      * Gets the overriding chronology to be used during formatting.
1444      * <p>
1445      * This returns the override chronology, used to convert dates.
1446      * By default, a formatter has no override chronology, returning null.
1447      * See {@link #withChronology(Chronology)} for more details on overriding.
1448      *
1449      * @return the override chronology of this formatter, null if no override
1450      */
1451     public Chronology getChronology() {
1452         return chrono;
1453     }
1454
1455     /**
1456      * Returns a copy of this formatter with a new override chronology.
1457      * <p>
1458      * This returns a formatter with similar state to this formatter but
1459      * with the override chronology set.
1460      * By default, a formatter has no override chronology, returning null.
1461      * <p>
1462      * If an override is added, then any date that is formatted or parsed will be affected.
1463      * <p>
1464      * When formatting, if the temporal object contains a date, then it will
1465      * be converted to a date in the override chronology.
1466      * Whether the temporal contains a date is determined by querying the
1467      * {@link ChronoField#EPOCH_DAY EPOCH_DAY} field.
1468      * Any time or zone will be retained unaltered unless overridden.
1469      * <p>
1470      * If the temporal object does not contain a date, but does contain one
1471      * or more {@code ChronoField} date fields, then a {@code DateTimeException}
1472      * is thrown. In all other cases, the override chronology is added to the temporal,
1473      * replacing any previous chronology, but without changing the date/time.
1474      * <p>
1475      * When parsing, there are two distinct cases to consider.
1476      * If a chronology has been parsed directly from the text, perhaps because
1477      * {@link DateTimeFormatterBuilder#appendChronologyId()} was used, then
1478      * this override chronology has no effect.
1479      * If no zone has been parsed, then this override chronology will be used
1480      * to interpret the {@code ChronoField} values into a date according to the
1481      * date resolving rules of the chronology.
1482      * <p>
1483      * This instance is immutable and unaffected by this method call.
1484      *
1485      * @param chrono the new chronology, null if no override
1486      * @return a formatter based on this formatter with the requested override chronology, not null
1487      */
1488     public DateTimeFormatter withChronology(Chronology chrono) {
1489         if (Objects.equals(this.chrono, chrono)) {
1490             return this;
1491         }

```

```

1492         return new DateTimeFormatter(printerParser, locale, decimalStyle, resolverStyle,
1493                                     resolverFields, chrono, zone);
1494     }
1495     //-----
1496     /**
1497      * Gets the overriding zone to be used during formatting.
1498      * <p>
1499      * This returns the override zone, used to convert instants.
1500      * By default, a formatter has no override zone, returning null.
1501      * See {@link #withZone(ZoneId)} for more details on overriding.
1502      *
1503      * @return the override zone of this formatter, null if no override
1504      */
1505     public ZoneId getZone() {
1506         return zone;
1507     }
1508
1509     /**
1510      * Returns a copy of this formatter with a new override zone.
1511      * <p>
1512      * This returns a formatter with similar state to this formatter but
1513      * with the override zone set.
1514      * By default, a formatter has no override zone, returning null.
1515      * <p>
1516      * If an override is added, then any instant that is formatted or parsed will be affected.
1517      * <p>
1518      * When formatting, if the temporal object contains an instant, then it will
1519      * be converted to a zoned date-time using the override zone.
1520      * Whether the temporal is an instant is determined by querying the
1521      * {@link ChronoField#INSTANT_SECONDS INSTANT_SECONDS} field.
1522      * If the input has a chronology then it will be retained unless overridden.
1523      * If the input does not have a chronology, such as {@code Instant}, then
1524      * the ISO chronology will be used.
1525      * <p>
1526      * If the temporal object does not contain an instant, but does contain
1527      * an offset then an additional check is made. If the normalized override
1528      * zone is an offset that differs from the offset of the temporal, then
1529      * a {@code DateTimeException} is thrown. In all other cases, the override
1530      * zone is added to the temporal, replacing any previous zone, but without
1531      * changing the date/time.
1532      * <p>
1533      * When parsing, there are two distinct cases to consider.
1534      * If a zone has been parsed directly from the text, perhaps because
1535      * {@link DateTimeFormatterBuilder#appendZoneId()} was used, then
1536      * this override zone has no effect.
1537      * If no zone has been parsed, then this override zone will be included in
1538      * the result of the parse where it can be used to build instants and date-times.
1539      * <p>
1540      * This instance is immutable and unaffected by this method call.
1541      *
1542      * @param zone the new override zone, null if no override
1543      * @return a formatter based on this formatter with the requested override zone, not null

```

```

1544     */
1545     public DateTimeFormatter withZone(ZoneId zone) {
1546         if (Objects.equals(this.zone, zone)) {
1547             return this;
1548         }
1549         return new DateTimeFormatter(printerParser, locale, decimalStyle, resolverStyle,
            resolverFields, chrono, zone);
1550     }
1551
1552     //-----
1553     /**
1554      * Gets the resolver style to use during parsing.
1555      * <p>
1556      * This returns the resolver style, used during the second phase of parsing
1557      * when fields are resolved into dates and times.
1558      * By default, a formatter has the {@link ResolverStyle#SMART SMART} resolver style.
1559      * See {@link #withResolverStyle(ResolverStyle)} for more details.
1560      *
1561      * @return the resolver style of this formatter, not null
1562      */
1563     public ResolverStyle getResolverStyle() {
1564         return resolverStyle;
1565     }
1566
1567     /**
1568      * Returns a copy of this formatter with a new resolver style.
1569      * <p>
1570      * This returns a formatter with similar state to this formatter but
1571      * with the resolver style set. By default, a formatter has the
1572      * {@link ResolverStyle#SMART SMART} resolver style.
1573      * <p>
1574      * Changing the resolver style only has an effect during parsing.
1575      * Parsing a text string occurs in two phases.
1576      * Phase 1 is a basic text parse according to the fields added to the builder.
1577      * Phase 2 resolves the parsed field-value pairs into date and/or time objects.
1578      * The resolver style is used to control how phase 2, resolving, happens.
1579      * See {@code ResolverStyle} for more information on the options available.
1580      * <p>
1581      * This instance is immutable and unaffected by this method call.
1582      *
1583      * @param resolverStyle the new resolver style, not null
1584      * @return a formatter based on this formatter with the requested resolver style, not null
1585      */
1586     public DateTimeFormatter withResolverStyle(ResolverStyle resolverStyle) {
1587         Objects.requireNonNull(resolverStyle, "resolverStyle");
1588         if (Objects.equals(this.resolverStyle, resolverStyle)) {
1589             return this;
1590         }
1591         return new DateTimeFormatter(printerParser, locale, decimalStyle, resolverStyle,
            resolverFields, chrono, zone);
1592     }
1593
1594     //-----

```

```
1595 /**
1596  * Gets the resolver fields to use during parsing.
1597  * <p>
1598  * This returns the resolver fields, used during the second phase of parsing
1599  * when fields are resolved into dates and times.
1600  * By default, a formatter has no resolver fields, and thus returns null.
1601  * See {@link #withResolverFields(Set)} for more details.
1602  *
1603  * @return the immutable set of resolver fields of this formatter, null if no fields
1604  */
1605 public Set<TemporalField> getResolverFields() {
1606     return resolverFields;
1607 }
1608
1609 /**
1610  * Returns a copy of this formatter with a new set of resolver fields.
1611  * <p>
1612  * This returns a formatter with similar state to this formatter but with
1613  * the resolver fields set. By default, a formatter has no resolver fields.
1614  * <p>
1615  * Changing the resolver fields only has an effect during parsing.
1616  * Parsing a text string occurs in two phases.
1617  * Phase 1 is a basic text parse according to the fields added to the builder.
1618  * Phase 2 resolves the parsed field-value pairs into date and/or time objects.
1619  * The resolver fields are used to filter the field-value pairs between phase 1 and 2.
1620  * <p>
1621  * This can be used to select between two or more ways that a date or time might
1622  * be resolved. For example, if the formatter consists of year, month, day-of-month
1623  * and day-of-year, then there are two ways to resolve a date.
1624  * Calling this method with the arguments {@link ChronoField#YEAR YEAR} and
1625  * {@link ChronoField#DAY_OF_YEAR DAY_OF_YEAR} will ensure that the date is
1626  * resolved using the year and day-of-year, effectively meaning that the month
1627  * and day-of-month are ignored during the resolving phase.
1628  * <p>
1629  * In a similar manner, this method can be used to ignore secondary fields that
1630  * would otherwise be cross-checked. For example, if the formatter consists of year,
1631  * month, day-of-month and day-of-week, then there is only one way to resolve a
1632  * date, but the parsed value for day-of-week will be cross-checked against the
1633  * resolved date. Calling this method with the arguments {@link ChronoField#YEAR YEAR},
1634  * {@link ChronoField#MONTH_OF_YEAR MONTH_OF_YEAR} and
1635  * {@link ChronoField#DAY_OF_MONTH DAY_OF_MONTH} will ensure that the date is
1636  * resolved correctly, but without any cross-check for the day-of-week.
1637  * <p>
1638  * In implementation terms, this method behaves as follows. The result of the
1639  * parsing phase can be considered to be a map of field to value. The behavior
1640  * of this method is to cause that map to be filtered between phase 1 and 2,
1641  * removing all fields other than those specified as arguments to this method.
1642  * <p>
1643  * This instance is immutable and unaffected by this method call.
1644  *
1645  * @param resolverFields the new set of resolver fields, null if no fields
1646  * @return a formatter based on this formatter with the requested resolver style, not null
1647  */
```

```

1648 public DateTimeFormatter withResolverFields(TemporalField... resolverFields) {
1649     Set<TemporalField> fields = null;
1650     if (resolverFields != null) {
1651         fields = Collections.unmodifiableSet(new HashSet<>(Arrays.asList(resolverFields)));
1652     }
1653     if (Objects.equals(this.resolverFields, fields)) {
1654         return this;
1655     }
1656     return new DateTimeFormatter(printerParser, locale, decimalStyle, resolverStyle, fields,
        chrono, zone);
1657 }
1658
1659 /**
1660  * Returns a copy of this formatter with a new set of resolver fields.
1661  * <p>
1662  * This returns a formatter with similar state to this formatter but with
1663  * the resolver fields set. By default, a formatter has no resolver fields.
1664  * <p>
1665  * Changing the resolver fields only has an effect during parsing.
1666  * Parsing a text string occurs in two phases.
1667  * Phase 1 is a basic text parse according to the fields added to the builder.
1668  * Phase 2 resolves the parsed field-value pairs into date and/or time objects.
1669  * The resolver fields are used to filter the field-value pairs between phase 1 and 2.
1670  * <p>
1671  * This can be used to select between two or more ways that a date or time might
1672  * be resolved. For example, if the formatter consists of year, month, day-of-month
1673  * and day-of-year, then there are two ways to resolve a date.
1674  * Calling this method with the arguments {@link ChronoField#YEAR YEAR} and
1675  * {@link ChronoField#DAY_OF_YEAR DAY_OF_YEAR} will ensure that the date is
1676  * resolved using the year and day-of-year, effectively meaning that the month
1677  * and day-of-month are ignored during the resolving phase.
1678  * <p>
1679  * In a similar manner, this method can be used to ignore secondary fields that
1680  * would otherwise be cross-checked. For example, if the formatter consists of year,
1681  * month, day-of-month and day-of-week, then there is only one way to resolve a
1682  * date, but the parsed value for day-of-week will be cross-checked against the
1683  * resolved date. Calling this method with the arguments {@link ChronoField#YEAR YEAR},
1684  * {@link ChronoField#MONTH_OF_YEAR MONTH_OF_YEAR} and
1685  * {@link ChronoField#DAY_OF_MONTH DAY_OF_MONTH} will ensure that the date is
1686  * resolved correctly, but without any cross-check for the day-of-week.
1687  * <p>
1688  * In implementation terms, this method behaves as follows. The result of the
1689  * parsing phase can be considered to be a map of field to value. The behavior
1690  * of this method is to cause that map to be filtered between phase 1 and 2,
1691  * removing all fields other than those specified as arguments to this method.
1692  * <p>
1693  * This instance is immutable and unaffected by this method call.
1694  *
1695  * @param resolverFields the new set of resolver fields, null if no fields
1696  * @return a formatter based on this formatter with the requested resolver style, not null
1697  */
1698 public DateTimeFormatter withResolverFields(Set<TemporalField> resolverFields) {
1699     if (Objects.equals(this.resolverFields, resolverFields)) {

```



```

1700         return this;
1701     }
1702     if (resolverFields != null) {
1703         resolverFields = Collections.unmodifiableSet(new HashSet<>(resolverFields));
1704     }
1705     return new DateTimeFormatter(printerParser, locale, decimalStyle, resolverStyle,
        resolverFields, chrono, zone);
1706 }
1707
1708 //-----
1709 /**
1710  * Formats a date-time object using this formatter.
1711  * <p>
1712  * This formats the date-time to a String using the rules of the formatter.
1713  *
1714  * @param temporal the temporal object to format, not null
1715  * @return the formatted string, not null
1716  * @throws DateTimeException if an error occurs during formatting
1717  */
1718 public String format(TemporalAccessor temporal) {
1719     StringBuilder buf = new StringBuilder(32);
1720     formatTo(temporal, buf);
1721     return buf.toString();
1722 }
1723
1724 //-----
1725 /**
1726  * Formats a date-time object to an {@code Appendable} using this formatter.
1727  * <p>
1728  * This outputs the formatted date-time to the specified destination.
1729  * {@link Appendable} is a general purpose interface that is implemented by all
1730  * key character output classes including {@code StringBuffer}, {@code StringBuilder},
1731  * {@code PrintStream} and {@code Writer}.
1732  * <p>
1733  * Although {@code Appendable} methods throw an {@code IOException}, this method does not.
1734  * Instead, any {@code IOException} is wrapped in a runtime exception.
1735  *
1736  * @param temporal the temporal object to format, not null
1737  * @param appendable the appendable to format to, not null
1738  * @throws DateTimeException if an error occurs during formatting
1739  */
1740 public void formatTo(TemporalAccessor temporal, Appendable appendable) {
1741     Objects.requireNonNull(temporal, "temporal");
1742     Objects.requireNonNull(appendable, "appendable");
1743     try {
1744         DateTimePrintContext context = new DateTimePrintContext(temporal, this);
1745         if (appendable instanceof StringBuilder) {
1746             printerParser.format(context, (StringBuilder) appendable);
1747         } else {
1748             // buffer output to avoid writing to appendable in case of error
1749             StringBuilder buf = new StringBuilder(32);
1750             printerParser.format(context, buf);
1751             appendable.append(buf);

```



```

1752     }
1753     } catch (IOException ex) {
1754         throw new DateTimeException(ex.getMessage(), ex);
1755     }
1756 }
1757
1758 //-----
1759 /**
1760  * Fully parses the text producing a temporal object.
1761  * <p>
1762  * This parses the entire text producing a temporal object.
1763  * It is typically more useful to use {@link #parse(CharSequence, TemporalQuery)}.
1764  * The result of this method is {@code TemporalAccessor} which has been resolved,
1765  * applying basic validation checks to help ensure a valid date-time.
1766  * <p>
1767  * If the parse completes without reading the entire length of the text,
1768  * or a problem occurs during parsing or merging, then an exception is thrown.
1769  *
1770  * @param text the text to parse, not null
1771  * @return the parsed temporal object, not null
1772  * @throws DateTimeParseException if unable to parse the requested result
1773  */
1774 public TemporalAccessor parse(CharSequence text) {
1775     Objects.requireNonNull(text, "text");
1776     try {
1777         return parseResolved0(text, null);
1778     } catch (DateTimeParseException ex) {
1779         throw ex;
1780     } catch (RuntimeException ex) {
1781         throw createError(text, ex);
1782     }
1783 }
1784
1785 /**
1786  * Parses the text using this formatter, providing control over the text position.
1787  * <p>
1788  * This parses the text without requiring the parse to start from the beginning
1789  * of the string or finish at the end.
1790  * The result of this method is {@code TemporalAccessor} which has been resolved,
1791  * applying basic validation checks to help ensure a valid date-time.
1792  * <p>
1793  * The text will be parsed from the specified start {@code ParsePosition}.
1794  * The entire length of the text does not have to be parsed, the {@code ParsePosition}
1795  * will be updated with the index at the end of parsing.
1796  * <p>
1797  * The operation of this method is slightly different to similar methods using
1798  * {@code ParsePosition} on {@code java.text.Format}. That class will return
1799  * errors using the error index on the {@code ParsePosition}. By contrast, this
1800  * method will throw a {@link DateTimeParseException} if an error occurs, with
1801  * the exception containing the error index.
1802  * This change in behavior is necessary due to the increased complexity of
1803  * parsing and resolving dates/times in this API.
1804  * <p>

```

```

1805     * If the formatter parses the same field more than once with different values,
1806     * the result will be an error.
1807     *
1808     * @param text the text to parse, not null
1809     * @param position the position to parse from, updated with length parsed
1810     * and the index of any error, not null
1811     * @return the parsed temporal object, not null
1812     * @throws DateTimeParseException if unable to parse the requested result
1813     * @throws IndexOutOfBoundsException if the position is invalid
1814     */
1815     public TemporalAccessor parse(CharSequence text, ParsePosition position) {
1816         Objects.requireNonNull(text, "text");
1817         Objects.requireNonNull(position, "position");
1818         try {
1819             return parseResolved0(text, position);
1820         } catch (DateTimeParseException | IndexOutOfBoundsException ex) {
1821             throw ex;
1822         } catch (RuntimeException ex) {
1823             throw createError(text, ex);
1824         }
1825     }
1826
1827     //-----
1828     /**
1829     * Fully parses the text producing an object of the specified type.
1830     * <p>
1831     * Most applications should use this method for parsing.
1832     * It parses the entire text to produce the required date-time.
1833     * The query is typically a method reference to a {@code from(TemporalAccessor)} method.
1834     * For example:
1835     * <pre>
1836     * LocalDateTime dt = parser.parse(str, LocalDateTime::from);
1837     * </pre>
1838     * If the parse completes without reading the entire length of the text,
1839     * or a problem occurs during parsing or merging, then an exception is thrown.
1840     *
1841     * @param <T> the type of the parsed date-time
1842     * @param text the text to parse, not null
1843     * @param query the query defining the type to parse to, not null
1844     * @return the parsed date-time, not null
1845     * @throws DateTimeParseException if unable to parse the requested result
1846     */
1847     public <T> T parse(CharSequence text, TemporalQuery<T> query) {
1848         Objects.requireNonNull(text, "text");
1849         Objects.requireNonNull(query, "query");
1850         try {
1851             return parseResolved0(text, null).query(query);
1852         } catch (DateTimeParseException ex) {
1853             throw ex;
1854         } catch (RuntimeException ex) {
1855             throw createError(text, ex);
1856         }
1857     }

```

```

1858
1859 /**
1860  * Fully parses the text producing an object of one of the specified types.
1861  * <p>
1862  * This parse method is convenient for use when the parser can handle optional elements.
1863  * For example, a pattern of 'uuuu-MM-dd HH.mm[ VV]' can be fully parsed to a {@code
1864      ZonedDateTime},
1865  * or partially parsed to a {@code LocalDateTime}.
1866  * The queries must be specified in order, starting from the best matching full-parse option
1867  * and ending with the worst matching minimal parse option.
1868  * The query is typically a method reference to a {@code from(TemporalAccessor)} method.
1869  * <p>
1870  * The result is associated with the first type that successfully parses.
1871  * Normally, applications will use {@code instanceof} to check the result.
1872  * For example:
1873  * <pre>
1874  * TemporalAccessor dt = parser.parseBest(str, ZonedDateTime::from, LocalDateTime::from);
1875  * if (dt instanceof ZonedDateTime) {
1876  *     ...
1877  * } else {
1878  *     ...
1879  * }
1880  * </pre>
1881  * If the parse completes without reading the entire length of the text,
1882  * or a problem occurs during parsing or merging, then an exception is thrown.
1883  *
1884  * @param text the text to parse, not null
1885  * @param queries the queries defining the types to attempt to parse to,
1886  * must implement {@code TemporalAccessor}, not null
1887  * @return the parsed date-time, not null
1888  * @throws IllegalArgumentException if less than 2 types are specified
1889  * @throws DateTimeParseException if unable to parse the requested result
1890  */
1891 public TemporalAccessor parseBest(CharSequence text, TemporalQuery<?>... queries) {
1892     Objects.requireNonNull(text, "text");
1893     Objects.requireNonNull(queries, "queries");
1894     if (queries.length < 2) {
1895         throw new IllegalArgumentException("At least two queries must be specified");
1896     }
1897     try {
1898         TemporalAccessor resolved = parseResolved0(text, null);
1899         for (TemporalQuery<?> query : queries) {
1900             try {
1901                 return (TemporalAccessor) resolved.query(query);
1902             } catch (RuntimeException ex) {
1903                 // continue
1904             }
1905         }
1906         throw new DateTimeException("Unable to convert parsed text using any of the specified
1907             queries");
1908     } catch (DateTimeParseException ex) {
1909         throw ex;
1910     } catch (RuntimeException ex) {

```

```

1909         throw createError(text, ex);
1910     }
1911 }
1912
1913 private DateTimeParseException createError(CharSequence text, RuntimeException ex) {
1914     String abbr;
1915     if (text.length() > 64) {
1916         abbr = text.subSequence(0, 64).toString() + "...";
1917     } else {
1918         abbr = text.toString();
1919     }
1920     return new DateTimeParseException("Text '" + abbr + "' could not be parsed: " + ex.
        getMessage(), text, 0, ex);
1921 }
1922
1923 //-----
1924 /**
1925  * Parses and resolves the specified text.
1926  * <p>
1927  * This parses to a {@code TemporalAccessor} ensuring that the text is fully parsed.
1928  *
1929  * @param text the text to parse, not null
1930  * @param position the position to parse from, updated with length parsed
1931  * and the index of any error, null if parsing whole string
1932  * @return the resolved result of the parse, not null
1933  * @throws DateTimeParseException if the parse fails
1934  * @throws DateTimeException if an error occurs while resolving the date or time
1935  * @throws IndexOutOfBoundsException if the position is invalid
1936  */
1937 private TemporalAccessor parseResolved0(final CharSequence text, final ParsePosition position)
    {
1938     ParsePosition pos = (position != null ? position : new ParsePosition(0));
1939     DateTimeParseContext context = parseUnresolved0(text, pos);
1940     if (context == null || pos.getErrorIndex() >= 0 || (position == null && pos.getIndex() <
        text.length())) {
1941         String abbr;
1942         if (text.length() > 64) {
1943             abbr = text.subSequence(0, 64).toString() + "...";
1944         } else {
1945             abbr = text.toString();
1946         }
1947         if (pos.getErrorIndex() >= 0) {
1948             throw new DateTimeParseException("Text '" + abbr + "' could not be parsed at index
                " +
1949                 pos.getErrorIndex(), text, pos.getErrorIndex());
1950         } else {
1951             throw new DateTimeParseException("Text '" + abbr + "' could not be parsed, unparsed
                text found at index " +
1952                 pos.getIndex(), text, pos.getIndex());
1953         }
1954     }
1955     return context.toResolved(resolverStyle, resolverFields);
1956 }

```

```

1957
1958 /**
1959  * Parses the text using this formatter, without resolving the result, intended
1960  * for advanced use cases.
1961  * <p>
1962  * Parsing is implemented as a two-phase operation.
1963  * First, the text is parsed using the layout defined by the formatter, producing
1964  * a {@code Map} of field to value, a {@code ZoneId} and a {@code Chronology}.
1965  * Second, the parsed data is <em>resolved</em>, by validating, combining and
1966  * simplifying the various fields into more useful ones.
1967  * This method performs the parsing stage but not the resolving stage.
1968  * <p>
1969  * The result of this method is {@code TemporalAccessor} which represents the
1970  * data as seen in the input. Values are not validated, thus parsing a date string
1971  * of '2012-00-65' would result in a temporal with three fields - year of '2012',
1972  * month of '0' and day-of-month of '65'.
1973  * <p>
1974  * The text will be parsed from the specified start {@code ParsePosition}.
1975  * The entire length of the text does not have to be parsed, the {@code ParsePosition}
1976  * will be updated with the index at the end of parsing.
1977  * <p>
1978  * Errors are returned using the error index field of the {@code ParsePosition}
1979  * instead of {@code DateTimeParseException}.
1980  * The returned error index will be set to an index indicative of the error.
1981  * Callers must check for errors before using the result.
1982  * <p>
1983  * If the formatter parses the same field more than once with different values,
1984  * the result will be an error.
1985  * <p>
1986  * This method is intended for advanced use cases that need access to the
1987  * internal state during parsing. Typical application code should use
1988  * {@link #parse(CharSequence, TemporalQuery)} or the parse method on the target type.
1989  *
1990  * @param text the text to parse, not null
1991  * @param position the position to parse from, updated with length parsed
1992  * and the index of any error, not null
1993  * @return the parsed text, null if the parse results in an error
1994  * @throws DateTimeException if some problem occurs during parsing
1995  * @throws IndexOutOfBoundsException if the position is invalid
1996  */
1997 public TemporalAccessor parseUnresolved(CharSequence text, ParsePosition position) {
1998     DateTimeParseContext context = parseUnresolved0(text, position);
1999     if (context == null) {
2000         return null;
2001     }
2002     return context.toUnresolved();
2003 }
2004
2005 private DateTimeParseContext parseUnresolved0(CharSequence text, ParsePosition position) {
2006     Objects.requireNonNull(text, "text");
2007     Objects.requireNonNull(position, "position");
2008     DateTimeParseContext context = new DateTimeParseContext(this);
2009     int pos = position.getIndex();

```

```

2010     pos = printerParser.parse(context, text, pos);
2011     if (pos < 0) {
2012         position.setErrorIndex(~pos); // index not updated from input
2013         return null;
2014     }
2015     position.setIndex(pos); // errorIndex not updated from input
2016     return context;
2017 }
2018
2019 //-----
2020 /**
2021  * Returns the formatter as a composite printer parser.
2022  *
2023  * @param optional whether the printer/parser should be optional
2024  * @return the printer/parser, not null
2025  */
2026 CompositePrinterParser toPrinterParser(boolean optional) {
2027     return printerParser.withOptional(optional);
2028 }
2029
2030 /**
2031  * Returns this formatter as a {@code java.text.Format} instance.
2032  * <p>
2033  * The returned {@link Format} instance will format any {@link TemporalAccessor}
2034  * and parses to a resolved {@link TemporalAccessor}.
2035  * <p>
2036  * Exceptions will follow the definitions of {@code Format}, see those methods
2037  * for details about {@code IllegalArgumentException} during formatting and
2038  * {@code ParseException} or null during parsing.
2039  * The format does not support attributing of the returned format string.
2040  *
2041  * @return this formatter as a classic format instance, not null
2042  */
2043 public Format toFormat() {
2044     return new ClassicFormat(this, null);
2045 }
2046
2047 /**
2048  * Returns this formatter as a {@code java.text.Format} instance that will
2049  * parse using the specified query.
2050  * <p>
2051  * The returned {@link Format} instance will format any {@link TemporalAccessor}
2052  * and parses to the type specified.
2053  * The type must be one that is supported by {@link #parse}.
2054  * <p>
2055  * Exceptions will follow the definitions of {@code Format}, see those methods
2056  * for details about {@code IllegalArgumentException} during formatting and
2057  * {@code ParseException} or null during parsing.
2058  * The format does not support attributing of the returned format string.
2059  *
2060  * @param parseQuery the query defining the type to parse to, not null
2061  * @return this formatter as a classic format instance, not null
2062  */

```

```

2063 public Format toFormat(TemporalQuery<?> parseQuery) {
2064     Objects.requireNonNull(parseQuery, "parseQuery");
2065     return new ClassicFormat(this, parseQuery);
2066 }
2067
2068 //-----
2069 /**
2070  * Returns a description of the underlying formatters.
2071  *
2072  * @return a description of this formatter, not null
2073  */
2074 @Override
2075 public String toString() {
2076     String pattern = printerParser.toString();
2077     pattern = pattern.startsWith("[") ? pattern : pattern.substring(1, pattern.length() - 1);
2078     return pattern;
2079     // TODO: Fix tests to not depend on toString()
2080     // return "DateTimeFormatter[" + locale +
2081     //         (chrono != null ? "," + chrono : "") +
2082     //         (zone != null ? "," + zone : "") +
2083     //         pattern + "]";
2084 }
2085
2086 //-----
2087 /**
2088  * Implements the classic Java Format API.
2089  * @serial exclude
2090  */
2091 @SuppressWarnings("serial") // not actually serializable
2092 static class ClassicFormat extends Format {
2093     /** The formatter. */
2094     private final DateTimeFormatter formatter;
2095     /** The type to be parsed. */
2096     private final TemporalQuery<?> parseType;
2097     /** Constructor. */
2098     public ClassicFormat(DateTimeFormatter formatter, TemporalQuery<?> parseType) {
2099         this.formatter = formatter;
2100         this.parseType = parseType;
2101     }
2102
2103     @Override
2104     public StringBuffer format(Object obj, StringBuffer toAppendTo, FieldPosition pos) {
2105         Objects.requireNonNull(obj, "obj");
2106         Objects.requireNonNull(toAppendTo, "toAppendTo");
2107         Objects.requireNonNull(pos, "pos");
2108         if (obj instanceof TemporalAccessor == false) {
2109             throw new IllegalArgumentException("Format target must implement TemporalAccessor");
2110         }
2111         pos.setBeginIndex(0);
2112         pos.setEndIndex(0);
2113         try {
2114             formatter.formatTo((TemporalAccessor) obj, toAppendTo);

```

```
2115         } catch (RuntimeException ex) {
2116             throw new IllegalArgumentException(ex.getMessage(), ex);
2117         }
2118         return toAppendTo;
2119     }
2120     @Override
2121     public Object parseObject(String text) throws ParseException {
2122         Objects.requireNonNull(text, "text");
2123         try {
2124             if (parseType == null) {
2125                 return formatter.parseResolved0(text, null);
2126             }
2127             return formatter.parse(text, parseType);
2128         } catch (DateTimeParseException ex) {
2129             throw new ParseException(ex.getMessage(), ex.getErrorIndex());
2130         } catch (RuntimeException ex) {
2131             throw (ParseException) new ParseException(ex.getMessage(), 0).initCause(ex);
2132         }
2133     }
2134     @Override
2135     public Object parseObject(String text, ParsePosition pos) {
2136         Objects.requireNonNull(text, "text");
2137         DateTimeParseContext context;
2138         try {
2139             context = formatter.parseUnresolved0(text, pos);
2140         } catch (IndexOutOfBoundsException ex) {
2141             if (pos.getErrorIndex() < 0) {
2142                 pos.setErrorIndex(0);
2143             }
2144             return null;
2145         }
2146         if (context == null) {
2147             if (pos.getErrorIndex() < 0) {
2148                 pos.setErrorIndex(0);
2149             }
2150             return null;
2151         }
2152         try {
2153             TemporalAccessor resolved = context.toResolved(formatter.resolverStyle, formatter.
2154                 resolverFields);
2155             if (parseType == null) {
2156                 return resolved;
2157             }
2158             return resolved.query(parseType);
2159         } catch (RuntimeException ex) {
2160             pos.setErrorIndex(0);
2161             return null;
2162         }
2163     }
2164 }
2165 }
```


6.2 DateTimeFormatterBuilder.java

Secuencia 99: java/time/format/DateTimeFormatterBuilder.java

```
1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2008-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
```

```
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.format;
63
64  import static java.time.temporal.ChronoField.DAY_OF_MONTH;
65  import static java.time.temporal.ChronoField.HOUR_OF_DAY;
66  import static java.time.temporal.ChronoField.INSTANT_SECONDS;
67  import static java.time.temporal.ChronoField.MINUTE_OF_HOUR;
68  import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
69  import static java.time.temporal.ChronoField.NANO_OF_SECOND;
70  import static java.time.temporal.ChronoField.OFFSET_SECONDS;
71  import static java.time.temporal.ChronoField.SECOND_OF_MINUTE;
72  import static java.time.temporal.ChronoField.YEAR;
73
74  import java.lang.ref.SoftReference;
75  import java.math.BigDecimal;
76  import java.math.BigInteger;
77  import java.math.RoundingMode;
78  import java.text.ParsePosition;
79  import java.time.DateTimeException;
80  import java.time.Instant;
81  import java.time.LocalDate;
82  import java.time.LocalDateTime;
83  import java.time.ZoneId;
84  import java.time.ZoneOffset;
85  import java.time.chrono.ChronoLocalDate;
86  import java.time.chrono.Chronology;
87  import java.time.chrono.IsoChronology;
88  import java.time.format.DateTimeTextProvider.LocaleStore;
89  import java.time.temporal.ChronoField;
90  import java.time.temporal.IsoFields;
91  import java.time.temporal.TemporalAccessor;
92  import java.time.temporal.TemporalField;
93  import java.time.temporal.TemporalQueries;
94  import java.time.temporal.TemporalQuery;
95  import java.time.temporal.ValueRange;
96  import java.time.temporal.WeekFields;
97  import java.time.zone.ZoneRulesProvider;
98  import java.util.AbstractMap.SimpleImmutableEntry;
99  import java.util.ArrayList;
100 import java.util.Arrays;
101 import java.util.Collections;
102 import java.util.Comparator;
103 import java.util.HashMap;
```

```
104 import java.util.HashSet;
105 import java.util.Iterator;
106 import java.util.LinkedHashMap;
107 import java.util.List;
108 import java.util.Locale;
109 import java.util.Map;
110 import java.util.Map.Entry;
111 import java.util.Objects;
112 import java.util.Set;
113 import java.util.TimeZone;
114 import java.util.concurrent.ConcurrentHashMap;
115 import java.util.concurrent.ConcurrentMap;
116
117 import sun.util.locale.provider.LocaleProviderAdapter;
118 import sun.util.locale.provider.LocaleResources;
119 import sun.util.locale.provider.TimeZoneNameUtility;
120
121 /**
122  * Builder to create date-time formatters.
123  * <p>
124  * This allows a {@code DateTimeFormatter} to be created.
125  * All date-time formatters are created ultimately using this builder.
126  * <p>
127  * The basic elements of date-time can all be added:
128  * <ul>
129  * <li>Value - a numeric value</li>
130  * <li>Fraction - a fractional value including the decimal place. Always use this when
131  * outputting fractions to ensure that the fraction is parsed correctly</li>
132  * <li>Text - the textual equivalent for the value</li>
133  * <li>OffsetId/Offset - the {@linkplain ZoneOffset zone offset}</li>
134  * <li>ZoneId - the {@linkplain ZoneId time-zone} id</li>
135  * <li>ZoneText - the name of the time-zone</li>
136  * <li>ChronologyId - the {@linkplain Chronology chronology} id</li>
137  * <li>ChronologyText - the name of the chronology</li>
138  * <li>Literal - a text literal</li>
139  * <li>Nested and Optional - formats can be nested or made optional</li>
140  * </ul>
141  * In addition, any of the elements may be decorated by padding, either with spaces or any other
142  * character.
143  * <p>
144  * Finally, a shorthand pattern, mostly compatible with {@code java.text.SimpleDateFormat
145  * SimpleDateFormat}
146  * can be used, see {@link #appendPattern(String)}.
147  * In practice, this simply parses the pattern and calls other methods on the builder.
148  *
149  * @implSpec
150  * This class is a mutable builder intended for use from a single thread.
151  *
152  * @since 1.8
153  */
154 public final class DateTimeFormatterBuilder {
```

```

155     * Query for a time-zone that is region-only.
156     */
157     private static final TemporalQuery<ZoneId> QUERY_REGION_ONLY = (temporal) -> {
158         ZoneId zone = temporal.query(TemporalQueries.zoneId());
159         return (zone != null && zone instanceof ZoneOffset == false ? zone : null);
160     };
161
162     /**
163     * The currently active builder, used by the outermost builder.
164     */
165     private DateTimeFormatterBuilder active = this;
166     /**
167     * The parent builder, null for the outermost builder.
168     */
169     private final DateTimeFormatterBuilder parent;
170     /**
171     * The list of printers that will be used.
172     */
173     private final List<DateTimePrinterParser> printerParsers = new ArrayList<>();
174     /**
175     * Whether this builder produces an optional formatter.
176     */
177     private final boolean optional;
178     /**
179     * The width to pad the next field to.
180     */
181     private int padNextWidth;
182     /**
183     * The character to pad the next field with.
184     */
185     private char padNextChar;
186     /**
187     * The index of the last variable width value parser.
188     */
189     private int valueParserIndex = -1;
190
191     /**
192     * Gets the formatting pattern for date and time styles for a locale and chronology.
193     * The locale and chronology are used to lookup the locale specific format
194     * for the requested dateStyle and/or timeStyle.
195     *
196     * @param dateStyle the FormatStyle for the date, null for time-only pattern
197     * @param timeStyle the FormatStyle for the time, null for date-only pattern
198     * @param chrono the Chronology, non-null
199     * @param locale the locale, non-null
200     * @return the locale and Chronology specific formatting pattern
201     * @throws IllegalArgumentException if both dateStyle and timeStyle are null
202     */
203     public static String getLocalizedDateTimePattern(FormatStyle dateStyle, FormatStyle timeStyle,
204         Chronology chrono, Locale locale) {
205         Objects.requireNonNull(locale, "locale");
206         Objects.requireNonNull(chrono, "chrono");
207         if (dateStyle == null && timeStyle == null) {

```

```

208         throw new IllegalArgumentException("Either dateStyle or timeStyle must be non-null");
209     }
210     LocaleResources lr = LocaleProviderAdapter.getResourceBundleBased().getLocaleResources(
211         locale);
212     String pattern = lr.getJavaTimeDateTimePattern(
213         convertStyle(timeStyle), convertStyle(dateStyle), chrono.getCalendarType());
214     return pattern;
215 }
216 /**
217  * Converts the given FormatStyle to the java.text.DateFormat style.
218  *
219  * @param style the FormatStyle style
220  * @return the int style, or -1 if style is null, indicating un-required
221  */
222 private static int convertStyle(FormatStyle style) {
223     if (style == null) {
224         return -1;
225     }
226     return style.ordinal(); // indices happen to align
227 }
228
229 /**
230  * Constructs a new instance of the builder.
231  */
232 public DateTimeFormatterBuilder() {
233     super();
234     parent = null;
235     optional = false;
236 }
237
238 /**
239  * Constructs a new instance of the builder.
240  *
241  * @param parent the parent builder, not null
242  * @param optional whether the formatter is optional, not null
243  */
244 private DateTimeFormatterBuilder(DateTimeFormatterBuilder parent, boolean optional) {
245     super();
246     this.parent = parent;
247     this.optional = optional;
248 }
249
250 //-----
251 /**
252  * Changes the parse style to be case sensitive for the remainder of the formatter.
253  * <p>
254  * Parsing can be case sensitive or insensitive - by default it is case sensitive.
255  * This method allows the case sensitivity setting of parsing to be changed.
256  * <p>
257  * Calling this method changes the state of the builder such that all
258  * subsequent builder method calls will parse text in case sensitive mode.
259  * See {@link #parseCaseInsensitive} for the opposite setting.

```

```

260     * The parse case sensitive/insensitive methods may be called at any point
261     * in the builder, thus the parser can swap between case parsing modes
262     * multiple times during the parse.
263     * <p>
264     * Since the default is case sensitive, this method should only be used after
265     * a previous call to {@code #parseCaseInsensitive}.
266     *
267     * @return this, for chaining, not null
268     */
269     public DateTimeFormatterBuilder parseCaseSensitive() {
270         appendInternal(SettingsParser.SENSITIVE);
271         return this;
272     }
273
274     /**
275     * Changes the parse style to be case insensitive for the remainder of the formatter.
276     * <p>
277     * Parsing can be case sensitive or insensitive - by default it is case sensitive.
278     * This method allows the case sensitivity setting of parsing to be changed.
279     * <p>
280     * Calling this method changes the state of the builder such that all
281     * subsequent builder method calls will parse text in case insensitive mode.
282     * See {@link #parseCaseSensitive()} for the opposite setting.
283     * The parse case sensitive/insensitive methods may be called at any point
284     * in the builder, thus the parser can swap between case parsing modes
285     * multiple times during the parse.
286     *
287     * @return this, for chaining, not null
288     */
289     public DateTimeFormatterBuilder parseCaseInsensitive() {
290         appendInternal(SettingsParser.INSENSITIVE);
291         return this;
292     }
293
294     //-----
295     /**
296     * Changes the parse style to be strict for the remainder of the formatter.
297     * <p>
298     * Parsing can be strict or lenient - by default its strict.
299     * This controls the degree of flexibility in matching the text and sign styles.
300     * <p>
301     * When used, this method changes the parsing to be strict from this point onwards.
302     * As strict is the default, this is normally only needed after calling {@link #parseLenient()}
303     * }.
304     * The change will remain in force until the end of the formatter that is eventually
305     * constructed or until {@code parseLenient} is called.
306     *
307     * @return this, for chaining, not null
308     */
309     public DateTimeFormatterBuilder parseStrict() {
310         appendInternal(SettingsParser.STRICT);
311         return this;
312     }

```

```

312
313 /**
314  * Changes the parse style to be lenient for the remainder of the formatter.
315  * Note that case sensitivity is set separately to this method.
316  * <p>
317  * Parsing can be strict or lenient - by default its strict.
318  * This controls the degree of flexibility in matching the text and sign styles.
319  * Applications calling this method should typically also call {@link #parseCaseInsensitive()}.
320  * <p>
321  * When used, this method changes the parsing to be lenient from this point onwards.
322  * The change will remain in force until the end of the formatter that is eventually
323  * constructed or until {@code parseStrict} is called.
324  *
325  * @return this, for chaining, not null
326  */
327 public DateTimeFormatterBuilder parseLenient() {
328     appendInternal(SettingsParser.LENIENT);
329     return this;
330 }
331
332 //-----
333 /**
334  * Appends a default value for a field to the formatter for use in parsing.
335  * <p>
336  * This appends an instruction to the builder to inject a default value
337  * into the parsed result. This is especially useful in conjunction with
338  * optional parts of the formatter.
339  * <p>
340  * For example, consider a formatter that parses the year, followed by
341  * an optional month, with a further optional day-of-month. Using such a
342  * formatter would require the calling code to check whether a full date,
343  * year-month or just a year had been parsed. This method can be used to
344  * default the month and day-of-month to a sensible value, such as the
345  * first of the month, allowing the calling code to always get a date.
346  * <p>
347  * During formatting, this method has no effect.
348  * <p>
349  * During parsing, the current state of the parse is inspected.
350  * If the specified field has no associated value, because it has not been
351  * parsed successfully at that point, then the specified value is injected
352  * into the parse result. Injection is immediate, thus the field-value pair
353  * will be visible to any subsequent elements in the formatter.
354  * As such, this method is normally called at the end of the builder.
355  *
356  * @param field the field to default the value of, not null
357  * @param value the value to default the field to
358  * @return this, for chaining, not null
359  */
360 public DateTimeFormatterBuilder parseDefaulting(TemporalField field, long value) {
361     Objects.requireNonNull(field, "field");
362     appendInternal(new DefaultValueParser(field, value));
363     return this;
364 }

```

```

365
366 //-----
367 /**
368  * Appends the value of a date-time field to the formatter using a normal
369  * output style.
370  * <p>
371  * The value of the field will be output during a format.
372  * If the value cannot be obtained then an exception will be thrown.
373  * <p>
374  * The value will be printed as per the normal format of an integer value.
375  * Only negative numbers will be signed. No padding will be added.
376  * <p>
377  * The parser for a variable width value such as this normally behaves greedily,
378  * requiring one digit, but accepting as many digits as possible.
379  * This behavior can be affected by 'adjacent value parsing'.
380  * See {@link #appendValue(java.time.temporal.TemporalField, int)} for full details.
381  *
382  * @param field the field to append, not null
383  * @return this, for chaining, not null
384  */
385 public DateTimeFormatterBuilder appendValue(TemporalField field) {
386     Objects.requireNonNull(field, "field");
387     appendValue(new NumberPrinterParser(field, 1, 19, SignStyle.NORMAL));
388     return this;
389 }
390
391 /**
392  * Appends the value of a date-time field to the formatter using a fixed
393  * width, zero-padded approach.
394  * <p>
395  * The value of the field will be output during a format.
396  * If the value cannot be obtained then an exception will be thrown.
397  * <p>
398  * The value will be zero-padded on the left. If the size of the value
399  * means that it cannot be printed within the width then an exception is thrown.
400  * If the value of the field is negative then an exception is thrown during formatting.
401  * <p>
402  * This method supports a special technique of parsing known as 'adjacent value parsing'.
403  * This technique solves the problem where a value, variable or fixed width, is followed by one
404     or more
405     fixed length values. The standard parser is greedy, and thus it would normally
406     steal the digits that are needed by the fixed width value parsers that follow the
407     variable width one.
408  * <p>
409  * No action is required to initiate 'adjacent value parsing'.
410  * When a call to {@code appendValue} is made, the builder
411  * enters adjacent value parsing setup mode. If the immediately subsequent method
412  * call or calls on the same builder are for a fixed width value, then the parser will reserve
413  * space so that the fixed width values can be parsed.
414  * <p>
415  * For example, consider {@code builder.appendValue(YEAR).appendValue(MONTH_OF_YEAR, 2);}
416  * The year is a variable width parse of between 1 and 19 digits.
417  * The month is a fixed width parse of 2 digits.

```



```

417 * Because these were appended to the same builder immediately after one another,
418 * the year parser will reserve two digits for the month to parse.
419 * Thus, the text '201106' will correctly parse to a year of 2011 and a month of 6.
420 * Without adjacent value parsing, the year would greedily parse all six digits and leave
421 * nothing for the month.
422 * <p>
423 * Adjacent value parsing applies to each set of fixed width not-negative values in the parser
424 * that immediately follow any kind of value, variable or fixed width.
425 * Calling any other append method will end the setup of adjacent value parsing.
426 * Thus, in the unlikely event that you need to avoid adjacent value parsing behavior,
427 * simply add the {@code appendValue} to another {@code DateTimeFormatterBuilder}
428 * and add that to this builder.
429 * <p>
430 * If adjacent parsing is active, then parsing must match exactly the specified
431 * number of digits in both strict and lenient modes.
432 * In addition, no positive or negative sign is permitted.
433 *
434 * @param field the field to append, not null
435 * @param width the width of the printed field, from 1 to 19
436 * @return this, for chaining, not null
437 * @throws IllegalArgumentException if the width is invalid
438 */
439 public DateTimeFormatterBuilder appendValue(TemporalField field, int width) {
440     Objects.requireNonNull(field, "field");
441     if (width < 1 || width > 19) {
442         throw new IllegalArgumentException("The width must be from 1 to 19 inclusive but was "
443             + width);
444     }
445     NumberPrinterParser pp = new NumberPrinterParser(field, width, width, SignStyle.
446         NOT_NEGATIVE);
447     appendValue(pp);
448     return this;
449 }
450 /**
451 * Appends the value of a date-time field to the formatter providing full
452 * control over formatting.
453 * <p>
454 * The value of the field will be output during a format.
455 * If the value cannot be obtained then an exception will be thrown.
456 * <p>
457 * This method provides full control of the numeric formatting, including
458 * zero-padding and the positive/negative sign.
459 * <p>
460 * The parser for a variable width value such as this normally behaves greedily,
461 * accepting as many digits as possible.
462 * This behavior can be affected by 'adjacent value parsing'.
463 * See {@link #appendValue(java.time.temporal.TemporalField, int)} for full details.
464 * <p>
465 * In strict parsing mode, the minimum number of parsed digits is {@code minWidth}
466 * and the maximum is {@code maxWidth}.
467 * In lenient parsing mode, the minimum number of parsed digits is one
468 * and the maximum is 19 (except as limited by adjacent value parsing).

```

```

468 * <p>
469 * If this method is invoked with equal minimum and maximum widths and a sign style of
470 * {@code NOT_NEGATIVE} then it delegates to {@code appendValue(TemporalField,int)}.
471 * In this scenario, the formatting and parsing behavior described there occur.
472 *
473 * @param field the field to append, not null
474 * @param minWidth the minimum field width of the printed field, from 1 to 19
475 * @param maxWidth the maximum field width of the printed field, from 1 to 19
476 * @param signStyle the positive/negative output style, not null
477 * @return this, for chaining, not null
478 * @throws IllegalArgumentException if the widths are invalid
479 */
480 public DateTimeFormatterBuilder appendValue(
481     TemporalField field, int minWidth, int maxWidth, SignStyle signStyle) {
482     if (minWidth == maxWidth && signStyle == SignStyle.NOT_NEGATIVE) {
483         return appendValue(field, maxWidth);
484     }
485     Objects.requireNonNull(field, "field");
486     Objects.requireNonNull(signStyle, "signStyle");
487     if (minWidth < 1 || minWidth > 19) {
488         throw new IllegalArgumentException("The minimum width must be from 1 to 19 inclusive
489             but was " + minWidth);
490     }
491     if (maxWidth < 1 || maxWidth > 19) {
492         throw new IllegalArgumentException("The maximum width must be from 1 to 19 inclusive
493             but was " + maxWidth);
494     }
495     if (maxWidth < minWidth) {
496         throw new IllegalArgumentException("The maximum width must exceed or equal the minimum
497             width but " +
498             maxWidth + " < " + minWidth);
499     }
500     NumberPrinterParser pp = new NumberPrinterParser(field, minWidth, maxWidth, signStyle);
501     appendValue(pp);
502     return this;
503 }
504
505 //-----
506 /**
507 * Appends the reduced value of a date-time field to the formatter.
508 * <p>
509 * Since fields such as year vary by chronology, it is recommended to use the
510 * {@link #appendValueReduced(TemporalField, int, int, ChronoLocalDate)} date}
511 * variant of this method in most cases. This variant is suitable for
512 * simple fields or working with only the ISO chronology.
513 * <p>
514 * For formatting, the {@code width} and {@code maxWidth} are used to
515 * determine the number of characters to format.
516 * If they are equal then the format is fixed width.
517 * If the value of the field is within the range of the {@code baseValue} using
518 * {@code width} characters then the reduced value is formatted otherwise the value is
519 * truncated to fit {@code maxWidth}.
520 * The rightmost characters are output to match the width, left padding with zero.

```

```

518 * <p>
519 * For strict parsing, the number of characters allowed by {@code width} to {@code maxWidth}
    are parsed.
520 * For lenient parsing, the number of characters must be at least 1 and less than 10.
521 * If the number of digits parsed is equal to {@code width} and the value is positive,
522 * the value of the field is computed to be the first number greater than
523 * or equal to the {@code baseValue} with the same least significant characters,
524 * otherwise the value parsed is the field value.
525 * This allows a reduced value to be entered for values in range of the baseValue
526 * and width and absolute values can be entered for values outside the range.
527 * <p>
528 * For example, a base value of {@code 1980} and a width of {@code 2} will have
529 * valid values from {@code 1980} to {@code 2079}.
530 * During parsing, the text {@code "12"} will result in the value {@code 2012} as that
531 * is the value within the range where the last two characters are "12".
532 * By contrast, parsing the text {@code "1915"} will result in the value {@code 1915}.
533 *
534 * @param field the field to append, not null
535 * @param width the field width of the printed and parsed field, from 1 to 10
536 * @param maxWidth the maximum field width of the printed field, from 1 to 10
537 * @param baseValue the base value of the range of valid values
538 * @return this, for chaining, not null
539 * @throws IllegalArgumentException if the width or base value is invalid
540 */
541 public DateTimeFormatterBuilder appendValueReduced(TemporalField field,
542     int width, int maxWidth, int baseValue) {
543     Objects.requireNonNull(field, "field");
544     ReducedPrinterParser pp = new ReducedPrinterParser(field, width, maxWidth, baseValue, null)
545     ;
546     appendValue(pp);
547     return this;
548 }
549 /**
550 * Appends the reduced value of a date-time field to the formatter.
551 * <p>
552 * This is typically used for formatting and parsing a two digit year.
553 * <p>
554 * The base date is used to calculate the full value during parsing.
555 * For example, if the base date is 1950-01-01 then parsed values for
556 * a two digit year parse will be in the range 1950-01-01 to 2049-12-31.
557 * Only the year would be extracted from the date, thus a base date of
558 * 1950-08-25 would also parse to the range 1950-01-01 to 2049-12-31.
559 * This behavior is necessary to support fields such as week-based-year
560 * or other calendar systems where the parsed value does not align with
561 * standard ISO years.
562 * <p>
563 * The exact behavior is as follows. Parse the full set of fields and
564 * determine the effective chronology using the last chronology if
565 * it appears more than once. Then convert the base date to the
566 * effective chronology. Then extract the specified field from the
567 * chronology-specific base date and use it to determine the
568 * {@code baseValue} used below.

```

```

569 * <p>
570 * For formatting, the {@code width} and {@code maxWidth} are used to
571 * determine the number of characters to format.
572 * If they are equal then the format is fixed width.
573 * If the value of the field is within the range of the {@code baseValue} using
574 * {@code width} characters then the reduced value is formatted otherwise the value is
575 * truncated to fit {@code maxWidth}.
576 * The rightmost characters are output to match the width, left padding with zero.
577 * <p>
578 * For strict parsing, the number of characters allowed by {@code width} to {@code maxWidth}
   are parsed.
579 * For lenient parsing, the number of characters must be at least 1 and less than 10.
580 * If the number of digits parsed is equal to {@code width} and the value is positive,
581 * the value of the field is computed to be the first number greater than
582 * or equal to the {@code baseValue} with the same least significant characters,
583 * otherwise the value parsed is the field value.
584 * This allows a reduced value to be entered for values in range of the baseValue
585 * and width and absolute values can be entered for values outside the range.
586 * <p>
587 * For example, a base value of {@code 1980} and a width of {@code 2} will have
588 * valid values from {@code 1980} to {@code 2079}.
589 * During parsing, the text {@code "12"} will result in the value {@code 2012} as that
590 * is the value within the range where the last two characters are "12".
591 * By contrast, parsing the text {@code "1915"} will result in the value {@code 1915}.
592 *
593 * @param field the field to append, not null
594 * @param width the field width of the printed and parsed field, from 1 to 10
595 * @param maxWidth the maximum field width of the printed field, from 1 to 10
596 * @param baseDate the base date used to calculate the base value for the range
597 * of valid values in the parsed chronology, not null
598 * @return this, for chaining, not null
599 * @throws IllegalArgumentException if the width or base value is invalid
600 */
601 public DateTimeFormatterBuilder appendValueReduced(
602     TemporalField field, int width, int maxWidth, ChronoLocalDate baseDate) {
603     Objects.requireNonNull(field, "field");
604     Objects.requireNonNull(baseDate, "baseDate");
605     ReducedPrinterParser pp = new ReducedPrinterParser(field, width, maxWidth, 0, baseDate);
606     appendValue(pp);
607     return this;
608 }
609
610 /**
611 * Appends a fixed or variable width printer-parser handling adjacent value mode.
612 * If a PrinterParser is not active then the new PrinterParser becomes
613 * the active PrinterParser.
614 * Otherwise, the active PrinterParser is modified depending on the new PrinterParser.
615 * If the new PrinterParser is fixed width and has sign style {@code NOT_NEGATIVE}
616 * then its width is added to the active PP and
617 * the new PrinterParser is forced to be fixed width.
618 * If the new PrinterParser is variable width, the active PrinterParser is changed
619 * to be fixed width and the new PrinterParser becomes the active PP.
620 *

```

```

621     * @param pp the printer-parser, not null
622     * @return this, for chaining, not null
623     */
624     private DateTimeFormatterBuilder appendValue(NumberPrinterParser pp) {
625         if (active.valueParserIndex >= 0) {
626             final int activeValueParser = active.valueParserIndex;
627
628             // adjacent parsing mode, update setting in previous parsers
629             NumberPrinterParser basePP = (NumberPrinterParser) active.printerParsers.get(
                activeValueParser);
630             if (pp.minWidth == pp.maxWidth && pp.signStyle == SignStyle.NOT_NEGATIVE) {
631                 // Append the width to the subsequentWidth of the active parser
632                 basePP = basePP.withSubsequentWidth(pp.maxWidth);
633                 // Append the new parser as a fixed width
634                 appendInternal(pp.withFixedWidth());
635                 // Retain the previous active parser
636                 active.valueParserIndex = activeValueParser;
637             } else {
638                 // Modify the active parser to be fixed width
639                 basePP = basePP.withFixedWidth();
640                 // The new parser becomes the new active parser
641                 active.valueParserIndex = appendInternal(pp);
642             }
643             // Replace the modified parser with the updated one
644             active.printerParsers.set(activeValueParser, basePP);
645         } else {
646             // The new Parser becomes the active parser
647             active.valueParserIndex = appendInternal(pp);
648         }
649         return this;
650     }
651
652     //-----
653     /**
654     * Appends the fractional value of a date-time field to the formatter.
655     * <p>
656     * The fractional value of the field will be output including the
657     * preceding decimal point. The preceding value is not output.
658     * For example, the second-of-minute value of 15 would be output as {@code .25}.
659     * <p>
660     * The width of the printed fraction can be controlled. Setting the
661     * minimum width to zero will cause no output to be generated.
662     * The printed fraction will have the minimum width necessary between
663     * the minimum and maximum widths - trailing zeroes are omitted.
664     * No rounding occurs due to the maximum width - digits are simply dropped.
665     * <p>
666     * When parsing in strict mode, the number of parsed digits must be between
667     * the minimum and maximum width. When parsing in lenient mode, the minimum
668     * width is considered to be zero and the maximum is nine.
669     * <p>
670     * If the value cannot be obtained then an exception will be thrown.
671     * If the value is negative an exception will be thrown.
672     * If the field does not have a fixed set of valid values then an

```

```

673     * exception will be thrown.
674     * If the field value in the date-time to be printed is invalid it
675     * cannot be printed and an exception will be thrown.
676     *
677     * @param field the field to append, not null
678     * @param minWidth the minimum width of the field excluding the decimal point, from 0 to 9
679     * @param maxWidth the maximum width of the field excluding the decimal point, from 1 to 9
680     * @param decimalPoint whether to output the localized decimal point symbol
681     * @return this, for chaining, not null
682     * @throws IllegalArgumentException if the field has a variable set of valid values or
683     * either width is invalid
684     */
685     public DateTimeFormatterBuilder appendFraction(
686         TemporalField field, int minWidth, int maxWidth, boolean decimalPoint) {
687         appendInternal(new FractionPrinterParser(field, minWidth, maxWidth, decimalPoint));
688         return this;
689     }
690
691     //-----
692     /**
693     * Appends the text of a date-time field to the formatter using the full
694     * text style.
695     * <p>
696     * The text of the field will be output during a format.
697     * The value must be within the valid range of the field.
698     * If the value cannot be obtained then an exception will be thrown.
699     * If the field has no textual representation, then the numeric value will be used.
700     * <p>
701     * The value will be printed as per the normal format of an integer value.
702     * Only negative numbers will be signed. No padding will be added.
703     *
704     * @param field the field to append, not null
705     * @return this, for chaining, not null
706     */
707     public DateTimeFormatterBuilder appendText(TemporalField field) {
708         return appendText(field, TextStyle.FULL);
709     }
710
711     /**
712     * Appends the text of a date-time field to the formatter.
713     * <p>
714     * The text of the field will be output during a format.
715     * The value must be within the valid range of the field.
716     * If the value cannot be obtained then an exception will be thrown.
717     * If the field has no textual representation, then the numeric value will be used.
718     * <p>
719     * The value will be printed as per the normal format of an integer value.
720     * Only negative numbers will be signed. No padding will be added.
721     *
722     * @param field the field to append, not null
723     * @param textStyle the text style to use, not null
724     * @return this, for chaining, not null
725     */

```

```

726 public DateTimeFormatterBuilder appendText(TemporalField field, TextStyle textStyle) {
727     Objects.requireNonNull(field, "field");
728     Objects.requireNonNull(textStyle, "textStyle");
729     appendInternal(new TextPrinterParser(field, textStyle, DateTimeTextProvider.getInstance()))
730     ;
731     return this;
732 }
733 /**
734  * Appends the text of a date-time field to the formatter using the specified
735  * map to supply the text.
736  * <p>
737  * The standard text outputting methods use the localized text in the JDK.
738  * This method allows that text to be specified directly.
739  * The supplied map is not validated by the builder to ensure that formatting or
740  * parsing is possible, thus an invalid map may throw an error during later use.
741  * <p>
742  * Supplying the map of text provides considerable flexibility in formatting and parsing.
743  * For example, a legacy application might require or supply the months of the
744  * year as "JNY", "FBY", "MCH" etc. These do not match the standard set of text
745  * for localized month names. Using this method, a map can be created which
746  * defines the connection between each value and the text:
747  * <pre>
748  * Map<Long, String> map = new HashMap<>();
749  * map.put(1L, "JNY");
750  * map.put(2L, "FBY");
751  * map.put(3L, "MCH");
752  * ...
753  * builder.appendText(MONTH_OF_YEAR, map);
754  * </pre>
755  * <p>
756  * Other uses might be to output the value with a suffix, such as "1st", "2nd", "3rd",
757  * or as Roman numerals "I", "II", "III", "IV".
758  * <p>
759  * During formatting, the value is obtained and checked that it is in the valid range.
760  * If text is not available for the value then it is output as a number.
761  * During parsing, the parser will match against the map of text and numeric values.
762  *
763  * @param field the field to append, not null
764  * @param textLookup the map from the value to the text
765  * @return this, for chaining, not null
766  */
767 public DateTimeFormatterBuilder appendText(TemporalField field, Map<Long, String> textLookup) {
768     Objects.requireNonNull(field, "field");
769     Objects.requireNonNull(textLookup, "textLookup");
770     Map<Long, String> copy = new LinkedHashMap<>(textLookup);
771     Map<TextStyle, Map<Long, String>> map = Collections.singletonMap(TextStyle.FULL, copy);
772     final LocaleStore store = new LocaleStore(map);
773     DateTimeTextProvider provider = new DateTimeTextProvider() {
774         @Override
775         public String getText(TemporalField field, long value, TextStyle style, Locale locale)
776         {
777             return store.getText(value, style);

```



```

777     }
778     @Override
779     public Iterator<Entry<String, Long>> getTextIterator(TemporalField field, TextStyle
        style, Locale locale) {
780         return store.getTextIterator(style);
781     }
782 };
783 appendInternal(new TextPrinterParser(field, TextStyle.FULL, provider));
784 return this;
785 }
786
787 //-----
788 /**
789  * Appends an instant using ISO-8601 to the formatter, formatting fractional
790  * digits in groups of three.
791  * <p>
792  * Instants have a fixed output format.
793  * They are converted to a date-time with a zone-offset of UTC and formatted
794  * using the standard ISO-8601 format.
795  * With this method, formatting nano-of-second outputs zero, three, six
796  * or nine digits as necessary.
797  * The localized decimal style is not used.
798  * <p>
799  * The instant is obtained using {@link ChronoField#INSTANT_SECONDS INSTANT_SECONDS}
800  * and optionally (@code NANO_OF_SECOND). The value of {@code INSTANT_SECONDS}
801  * may be outside the maximum range of {@code LocalDateTime}.
802  * <p>
803  * The {@linkplain ResolverStyle resolver style} has no effect on instant parsing.
804  * The end-of-day time of '24:00' is handled as midnight at the start of the following day.
805  * The leap-second time of '23:59:59' is handled to some degree, see
806  * {@link DateTimeFormatter#parsedLeapSecond()} for full details.
807  * <p>
808  * An alternative to this method is to format/parse the instant as a single
809  * epoch-seconds value. That is achieved using {@code appendValue(INSTANT_SECONDS)}.
810  *
811  * @return this, for chaining, not null
812  */
813 public DateTimeFormatterBuilder appendInstant() {
814     appendInternal(new InstantPrinterParser(-2));
815     return this;
816 }
817
818 /**
819  * Appends an instant using ISO-8601 to the formatter with control over
820  * the number of fractional digits.
821  * <p>
822  * Instants have a fixed output format, although this method provides some
823  * control over the fractional digits. They are converted to a date-time
824  * with a zone-offset of UTC and printed using the standard ISO-8601 format.
825  * The localized decimal style is not used.
826  * <p>
827  * The {@code fractionalDigits} parameter allows the output of the fractional
828  * second to be controlled. Specifying zero will cause no fractional digits

```



```

829 * to be output. From 1 to 9 will output an increasing number of digits, using
830 * zero right-padding if necessary. The special value -1 is used to output as
831 * many digits as necessary to avoid any trailing zeroes.
832 * <p>
833 * When parsing in strict mode, the number of parsed digits must match the
834 * fractional digits. When parsing in lenient mode, any number of fractional
835 * digits from zero to nine are accepted.
836 * <p>
837 * The instant is obtained using {@link ChronoField#INSTANT_SECONDS INSTANT_SECONDS}
838 * and optionally (@code NANO_OF_SECOND). The value of {@code INSTANT_SECONDS}
839 * may be outside the maximum range of {@code LocalDateTime}.
840 * <p>
841 * The {@linkplain ResolverStyle resolver style} has no effect on instant parsing.
842 * The end-of-day time of '24:00' is handled as midnight at the start of the following day.
843 * The leap-second time of '23:59:60' is handled to some degree, see
844 * {@link DateTimeFormatter#parsedLeapSecond()} for full details.
845 * <p>
846 * An alternative to this method is to format/parse the instant as a single
847 * epoch-seconds value. That is achieved using {@code appendValue(INSTANT_SECONDS)}.
848 *
849 * @param fractionalDigits the number of fractional second digits to format with,
850 * from 0 to 9, or -1 to use as many digits as necessary
851 * @return this, for chaining, not null
852 */
853 public DateTimeFormatterBuilder appendInstant(int fractionalDigits) {
854     if (fractionalDigits < -1 || fractionalDigits > 9) {
855         throw new IllegalArgumentException("The fractional digits must be from -1 to 9
856             inclusive but was " + fractionalDigits);
857     }
858     appendInternal(new InstantPrinterParser(fractionalDigits));
859     return this;
860 }
861
862 //-----
863 /**
864  * Appends the zone offset, such as '+01:00', to the formatter.
865  * <p>
866  * This appends an instruction to format/parse the offset ID to the builder.
867  * This is equivalent to calling {@code appendOffset("+HH:MM:ss", "Z")}.
868  *
869  * @return this, for chaining, not null
870  */
871 public DateTimeFormatterBuilder appendOffsetId() {
872     appendInternal(OffsetIdPrinterParser.INSTANCE_ID_Z);
873     return this;
874 }
875
876 /**
877  * Appends the zone offset, such as '+01:00', to the formatter.
878  * <p>
879  * This appends an instruction to format/parse the offset ID to the builder.
880  * <p>
881  * During formatting, the offset is obtained using a mechanism equivalent

```

```

881 * to querying the temporal with {@link TemporalQueries#offset()}.
882 * It will be printed using the format defined below.
883 * If the offset cannot be obtained then an exception is thrown unless the
884 * section of the formatter is optional.
885 * <p>
886 * During parsing, the offset is parsed using the format defined below.
887 * If the offset cannot be parsed then an exception is thrown unless the
888 * section of the formatter is optional.
889 * <p>
890 * The format of the offset is controlled by a pattern which must be one
891 * of the following:
892 * <ul>
893 * <li>{@code +HH} - hour only, ignoring minute and second
894 * <li>{@code +HHmm} - hour, with minute if non-zero, ignoring second, no colon
895 * <li>{@code +HH:mm} - hour, with minute if non-zero, ignoring second, with colon
896 * <li>{@code +HHMM} - hour and minute, ignoring second, no colon
897 * <li>{@code +HH:MM} - hour and minute, ignoring second, with colon
898 * <li>{@code +HHMMss} - hour and minute, with second if non-zero, no colon
899 * <li>{@code +HH:MM:ss} - hour and minute, with second if non-zero, with colon
900 * <li>{@code +HHMMSS} - hour, minute and second, no colon
901 * <li>{@code +HH:MM:SS} - hour, minute and second, with colon
902 * </ul>
903 * The "no offset" text controls what text is printed when the total amount of
904 * the offset fields to be output is zero.
905 * Example values would be 'Z', '+00:00', 'UTC' or 'GMT'.
906 * Three formats are accepted for parsing UTC - the "no offset" text, and the
907 * plus and minus versions of zero defined by the pattern.
908 *
909 * @param pattern the pattern to use, not null
910 * @param noOffsetText the text to use when the offset is zero, not null
911 * @return this, for chaining, not null
912 */
913 public DateTimeFormatterBuilder appendOffset(String pattern, String noOffsetText) {
914     appendInternal(new OffsetIdPrinterParser(pattern, noOffsetText));
915     return this;
916 }
917
918 /**
919 * Appends the localized zone offset, such as 'GMT+01:00', to the formatter.
920 * <p>
921 * This appends a localized zone offset to the builder, the format of the
922 * localized offset is controlled by the specified {@link FormatStyle style}
923 * to this method:
924 * <ul>
925 * <li>{@link TextStyle#FULL full} - formats with localized offset text, such
926 * as 'GMT', 2-digit hour and minute field, optional second field if non-zero,
927 * and colon.
928 * <li>{@link TextStyle#SHORT short} - formats with localized offset text,
929 * such as 'GMT', hour without leading zero, optional 2-digit minute and
930 * second if non-zero, and colon.
931 * </ul>
932 * <p>
933 * During formatting, the offset is obtained using a mechanism equivalent

```

```

934 * to querying the temporal with {@link TemporalQueries#offset()}.
935 * If the offset cannot be obtained then an exception is thrown unless the
936 * section of the formatter is optional.
937 * <p>
938 * During parsing, the offset is parsed using the format defined above.
939 * If the offset cannot be parsed then an exception is thrown unless the
940 * section of the formatter is optional.
941 * <p>
942 * @param style the format style to use, not null
943 * @return this, for chaining, not null
944 * @throws IllegalArgumentException if style is neither {@link TextStyle#FULL
945 * full} nor {@link TextStyle#SHORT short}
946 */
947 public DateTimeFormatterBuilder appendLocalizedOffset(TextStyle style) {
948     Objects.requireNonNull(style, "style");
949     if (style != TextStyle.FULL && style != TextStyle.SHORT) {
950         throw new IllegalArgumentException("Style must be either full or short");
951     }
952     appendInternal(new LocalizedOffsetIdPrinterParser(style));
953     return this;
954 }
955
956 //-----
957 /**
958 * Appends the time-zone ID, such as 'Europe/Paris' or '+02:00', to the formatter.
959 * <p>
960 * This appends an instruction to format/parse the zone ID to the builder.
961 * The zone ID is obtained in a strict manner suitable for {@code ZonedDateTime}.
962 * By contrast, {@code OffsetDateTime} does not have a zone ID suitable
963 * for use with this method, see {@link #appendZoneOrOffsetId()}.
964 * <p>
965 * During formatting, the zone is obtained using a mechanism equivalent
966 * to querying the temporal with {@link TemporalQueries#zoneId()}.
967 * It will be printed using the result of {@link ZoneId#getId()}.
968 * If the zone cannot be obtained then an exception is thrown unless the
969 * section of the formatter is optional.
970 * <p>
971 * During parsing, the text must match a known zone or offset.
972 * There are two types of zone ID, offset-based, such as '+01:30' and
973 * region-based, such as 'Europe/London'. These are parsed differently.
974 * If the parse starts with '+', '-', 'UT', 'UTC' or 'GMT', then the parser
975 * expects an offset-based zone and will not match region-based zones.
976 * The offset ID, such as '+02:30', may be at the start of the parse,
977 * or prefixed by 'UT', 'UTC' or 'GMT'. The offset ID parsing is
978 * equivalent to using {@link #appendOffset(String, String)} using the
979 * arguments 'HH:MM:ss' and the no offset string '0'.
980 * If the parse starts with 'UT', 'UTC' or 'GMT', and the parser cannot
981 * match a following offset ID, then {@link ZoneOffset#UTC} is selected.
982 * In all other cases, the list of known region-based zones is used to
983 * find the longest available match. If no match is found, and the parse
984 * starts with 'Z', then {@code ZoneOffset.UTC} is selected.
985 * The parser uses the {@link plain #parseCaseInsensitive()} case sensitive} setting.
986 * <p>

```

```

987 * For example, the following will parse:
988 * <pre>
989 * "Europe/London"      -- ZoneId.of("Europe/London")
990 * "Z"                  -- ZoneOffset.UTC
991 * "UT"                 -- ZoneId.of("UT")
992 * "UTC"                -- ZoneId.of("UTC")
993 * "GMT"                -- ZoneId.of("GMT")
994 * "+01:30"             -- ZoneOffset.of("+01:30")
995 * "UT+01:30"           -- ZoneOffset.of("+01:30")
996 * "UTC+01:30"          -- ZoneOffset.of("+01:30")
997 * "GMT+01:30"          -- ZoneOffset.of("+01:30")
998 * </pre>
999 *
1000 * @return this, for chaining, not null
1001 * @see #appendZoneRegionId()
1002 */
1003 public DateTimeFormatterBuilder appendZoneId() {
1004     appendInternal(new ZoneIdPrinterParser(TemporalQueries.zoneId(), "ZoneId()"));
1005     return this;
1006 }
1007
1008 /**
1009  * Appends the time-zone region ID, such as 'Europe/Paris', to the formatter,
1010  * rejecting the zone ID if it is a {@code ZoneOffset}.
1011  * <p>
1012  * This appends an instruction to format/parse the zone ID to the builder
1013  * only if it is a region-based ID.
1014  * <p>
1015  * During formatting, the zone is obtained using a mechanism equivalent
1016  * to querying the temporal with {@link TemporalQueries#zoneId()}.
1017  * If the zone is a {@code ZoneOffset} or it cannot be obtained then
1018  * an exception is thrown unless the section of the formatter is optional.
1019  * If the zone is not an offset, then the zone will be printed using
1020  * the zone ID from {@link ZoneId#getId()}.
1021  * <p>
1022  * During parsing, the text must match a known zone or offset.
1023  * There are two types of zone ID, offset-based, such as '+01:30' and
1024  * region-based, such as 'Europe/London'. These are parsed differently.
1025  * If the parse starts with '+', '-', 'UT', 'UTC' or 'GMT', then the parser
1026  * expects an offset-based zone and will not match region-based zones.
1027  * The offset ID, such as '+02:30', may be at the start of the parse,
1028  * or prefixed by 'UT', 'UTC' or 'GMT'. The offset ID parsing is
1029  * equivalent to using {@link #appendOffset(String, String)} using the
1030  * arguments 'HH:MM:ss' and the no offset string '0'.
1031  * If the parse starts with 'UT', 'UTC' or 'GMT', and the parser cannot
1032  * match a following offset ID, then {@link ZoneOffset#UTC} is selected.
1033  * In all other cases, the list of known region-based zones is used to
1034  * find the longest available match. If no match is found, and the parse
1035  * starts with 'Z', then {@code ZoneOffset.UTC} is selected.
1036  * The parser uses the {@linkplain #parseCaseInsensitive() case sensitive} setting.
1037  * <p>
1038  * For example, the following will parse:
1039  * <pre>

```

```

1040 * "Europe/London"      -- ZoneId.of("Europe/London")
1041 * "Z"                  -- ZoneOffset.UTC
1042 * "UT"                 -- ZoneId.of("UT")
1043 * "UTC"                -- ZoneId.of("UTC")
1044 * "GMT"                -- ZoneId.of("GMT")
1045 * "+01:30"             -- ZoneOffset.of("+01:30")
1046 * "UT+01:30"           -- ZoneOffset.of("+01:30")
1047 * "UTC+01:30"          -- ZoneOffset.of("+01:30")
1048 * "GMT+01:30"          -- ZoneOffset.of("+01:30")
1049 * </pre>
1050 * <p>
1051 * Note that this method is identical to {@code appendZoneId()} except
1052 * in the mechanism used to obtain the zone.
1053 * Note also that parsing accepts offsets, whereas formatting will never
1054 * produce one.
1055 *
1056 * @return this, for chaining, not null
1057 * @see #appendZoneId()
1058 */
1059 public DateTimeFormatterBuilder appendZoneRegionId() {
1060     appendInternal(new ZoneIdPrinterParser(QUERY_REGION_ONLY, "ZoneRegionId()"));
1061     return this;
1062 }
1063
1064 /**
1065 * Appends the time-zone ID, such as 'Europe/Paris' or '+02:00', to
1066 * the formatter, using the best available zone ID.
1067 * <p>
1068 * This appends an instruction to format/parse the best available
1069 * zone or offset ID to the builder.
1070 * The zone ID is obtained in a lenient manner that first attempts to
1071 * find a true zone ID, such as that on {@code ZonedDateTime}, and
1072 * then attempts to find an offset, such as that on {@code OffsetDateTime}.
1073 * <p>
1074 * During formatting, the zone is obtained using a mechanism equivalent
1075 * to querying the temporal with {@link TemporalQueries#zone()}.
1076 * It will be printed using the result of {@link ZoneId#getId()}.
1077 * If the zone cannot be obtained then an exception is thrown unless the
1078 * section of the formatter is optional.
1079 * <p>
1080 * During parsing, the text must match a known zone or offset.
1081 * There are two types of zone ID, offset-based, such as '+01:30' and
1082 * region-based, such as 'Europe/London'. These are parsed differently.
1083 * If the parse starts with '+', '-', 'UT', 'UTC' or 'GMT', then the parser
1084 * expects an offset-based zone and will not match region-based zones.
1085 * The offset ID, such as '+02:30', may be at the start of the parse,
1086 * or prefixed by 'UT', 'UTC' or 'GMT'. The offset ID parsing is
1087 * equivalent to using {@link #appendOffset(String, String)} using the
1088 * arguments 'HH:MM:ss' and the no offset string '0'.
1089 * If the parse starts with 'UT', 'UTC' or 'GMT', and the parser cannot
1090 * match a following offset ID, then {@link ZoneOffset#UTC} is selected.
1091 * In all other cases, the list of known region-based zones is used to
1092 * find the longest available match. If no match is found, and the parse

```

```

1093 * starts with 'Z', then {@code ZoneOffset.UTC} is selected.
1094 * The parser uses the {@linkplain #parseCaseInsensitive()} case sensitive} setting.
1095 * <p>
1096 * For example, the following will parse:
1097 * <pre>
1098 * "Europe/London"      -- ZoneId.of("Europe/London")
1099 * "Z"                  -- ZoneOffset.UTC
1100 * "UT"                 -- ZoneId.of("UT")
1101 * "UTC"                -- ZoneId.of("UTC")
1102 * "GMT"                -- ZoneId.of("GMT")
1103 * "+01:30"            -- ZoneOffset.of("+01:30")
1104 * "UT+01:30"          -- ZoneOffset.of("UT+01:30")
1105 * "UTC+01:30"         -- ZoneOffset.of("UTC+01:30")
1106 * "GMT+01:30"         -- ZoneOffset.of("GMT+01:30")
1107 * </pre>
1108 * <p>
1109 * Note that this method is identical to {@code appendZoneId()} except
1110 * in the mechanism used to obtain the zone.
1111 *
1112 * @return this, for chaining, not null
1113 * @see #appendZoneId()
1114 */
1115 public DateTimeFormatterBuilder appendZoneOrOffsetId() {
1116     appendInternal(new ZoneIdPrinterParser(TemporalQueries.zone(), "ZoneOrOffsetId("));
1117     return this;
1118 }
1119
1120 /**
1121 * Appends the time-zone name, such as 'British Summer Time', to the formatter.
1122 * <p>
1123 * This appends an instruction to format/parse the textual name of the zone to
1124 * the builder.
1125 * <p>
1126 * During formatting, the zone is obtained using a mechanism equivalent
1127 * to querying the temporal with {@link TemporalQueries#zoneId()}.
1128 * If the zone is a {@code ZoneOffset} it will be printed using the
1129 * result of {@link ZoneOffset#getId()}.
1130 * If the zone is not an offset, the textual name will be looked up
1131 * for the locale set in the {@link DateTimeFormatter}.
1132 * If the temporal object being printed represents an instant, then the text
1133 * will be the summer or winter time text as appropriate.
1134 * If the lookup for text does not find any suitable result, then the
1135 * {@link ZoneId#getId() ID} will be printed instead.
1136 * If the zone cannot be obtained then an exception is thrown unless the
1137 * section of the formatter is optional.
1138 * <p>
1139 * During parsing, either the textual zone name, the zone ID or the offset
1140 * is accepted. Many textual zone names are not unique, such as CST can be
1141 * for both "Central Standard Time" and "China Standard Time". In this
1142 * situation, the zone id will be determined by the region information from
1143 * formatter's {@link DateTimeFormatter#getLocale() locale} and the standard
1144 * zone id for that area, for example, America/New_York for the America Eastern
1145 * zone. The {@link #appendZoneText(TextStyle, Set)} may be used

```

```

1146     * to specify a set of preferred {@link ZoneId} in this situation.
1147     *
1148     * @param textStyle the text style to use, not null
1149     * @return this, for chaining, not null
1150     */
1151     public DateTimeFormatterBuilder appendZoneText(TextStyle textStyle) {
1152         appendInternal(new ZoneTextPrinterParser(textStyle, null));
1153         return this;
1154     }
1155
1156     /**
1157     * Appends the time-zone name, such as 'British Summer Time', to the formatter.
1158     * <p>
1159     * This appends an instruction to format/parse the textual name of the zone to
1160     * the builder.
1161     * <p>
1162     * During formatting, the zone is obtained using a mechanism equivalent
1163     * to querying the temporal with {@link TemporalQueries#zoneId()}.
1164     * If the zone is a {@code ZoneOffset} it will be printed using the
1165     * result of {@link ZoneOffset#getId()}.
1166     * If the zone is not an offset, the textual name will be looked up
1167     * for the locale set in the {@link DateTimeFormatter}.
1168     * If the temporal object being printed represents an instant, then the text
1169     * will be the summer or winter time text as appropriate.
1170     * If the lookup for text does not find any suitable result, then the
1171     * {@link ZoneId#getId() ID} will be printed instead.
1172     * If the zone cannot be obtained then an exception is thrown unless the
1173     * section of the formatter is optional.
1174     * <p>
1175     * During parsing, either the textual zone name, the zone ID or the offset
1176     * is accepted. Many textual zone names are not unique, such as CST can be
1177     * for both "Central Standard Time" and "China Standard Time". In this
1178     * situation, the zone id will be determined by the region information from
1179     * formatter's {@link DateTimeFormatter#getLocale() locale} and the standard
1180     * zone id for that area, for example, America/New_York for the America Eastern
1181     * zone. This method also allows a set of preferred {@link ZoneId} to be
1182     * specified for parsing. The matched preferred zone id will be used if the
1183     * textual zone name being parsed is not unique.
1184     * <p>
1185     * If the zone cannot be parsed then an exception is thrown unless the
1186     * section of the formatter is optional.
1187     *
1188     * @param textStyle the text style to use, not null
1189     * @param preferredZones the set of preferred zone ids, not null
1190     * @return this, for chaining, not null
1191     */
1192     public DateTimeFormatterBuilder appendZoneText(TextStyle textStyle,
1193                                                    Set<ZoneId> preferredZones) {
1194         Objects.requireNonNull(preferredZones, "preferredZones");
1195         appendInternal(new ZoneTextPrinterParser(textStyle, preferredZones));
1196         return this;
1197     }
1198

```



```

1199 //-----
1200 /**
1201  * Appends the chronology ID, such as 'ISO' or 'ThaiBuddhist', to the formatter.
1202  * <p>
1203  * This appends an instruction to format/parse the chronology ID to the builder.
1204  * <p>
1205  * During formatting, the chronology is obtained using a mechanism equivalent
1206  * to querying the temporal with {@link TemporalQueries#chronology()}.
1207  * It will be printed using the result of {@link Chronology#getId()}.
1208  * If the chronology cannot be obtained then an exception is thrown unless the
1209  * section of the formatter is optional.
1210  * <p>
1211  * During parsing, the chronology is parsed and must match one of the chronologies
1212  * in {@link Chronology#getAvailableChronologies()}.
1213  * If the chronology cannot be parsed then an exception is thrown unless the
1214  * section of the formatter is optional.
1215  * The parser uses the {@link PlainTextFormatter#parseCaseInsensitive() case sensitive} setting.
1216  *
1217  * @return this, for chaining, not null
1218  */
1219 public DateTimeFormatterBuilder appendChronologyId() {
1220     appendInternal(new ChronoPrinterParser(null));
1221     return this;
1222 }
1223
1224 /**
1225  * Appends the chronology name to the formatter.
1226  * <p>
1227  * The calendar system name will be output during a format.
1228  * If the chronology cannot be obtained then an exception will be thrown.
1229  *
1230  * @param textStyle the text style to use, not null
1231  * @return this, for chaining, not null
1232  */
1233 public DateTimeFormatterBuilder appendChronologyText(TextStyle textStyle) {
1234     Objects.requireNonNull(textStyle, "textStyle");
1235     appendInternal(new ChronoPrinterParser(textStyle));
1236     return this;
1237 }
1238
1239 //-----
1240 /**
1241  * Appends a localized date-time pattern to the formatter.
1242  * <p>
1243  * This appends a localized section to the builder, suitable for outputting
1244  * a date, time or date-time combination. The format of the localized
1245  * section is lazily looked up based on four items:
1246  * <ul>
1247  * <li>the {@code dateStyle} specified to this method
1248  * <li>the {@code timeStyle} specified to this method
1249  * <li>the {@code Locale} of the {@code DateTimeFormatter}
1250  * <li>the {@code Chronology}, selecting the best available
1251  * </ul>

```



```

1252     * During formatting, the chronology is obtained from the temporal object
1253     * being formatted, which may have been overridden by
1254     * {@link DateTimeFormatter#withChronology(Chronology)}.
1255     * <p>
1256     * During parsing, if a chronology has already been parsed, then it is used.
1257     * Otherwise the default from {@code DateTimeFormatter.withChronology(Chronology)}
1258     * is used, with {@code IsoChronology} as the fallback.
1259     * <p>
1260     * Note that this method provides similar functionality to methods on
1261     * {@code DateFormat} such as {@link java.text.DateFormat#getInstance(int, int)}.
1262     *
1263     * @param dateStyle the date style to use, null means no date required
1264     * @param timeStyle the time style to use, null means no time required
1265     * @return this, for chaining, not null
1266     * @throws IllegalArgumentException if both the date and time styles are null
1267     */
1268     public DateTimeFormatterBuilder appendLocalized(FormatStyle dateStyle, FormatStyle timeStyle) {
1269         if (dateStyle == null && timeStyle == null) {
1270             throw new IllegalArgumentException("Either the date or time style must be non-null");
1271         }
1272         appendInternal(new LocalizedPrinterParser(dateStyle, timeStyle));
1273         return this;
1274     }
1275
1276     //-----
1277     /**
1278     * Appends a character literal to the formatter.
1279     * <p>
1280     * This character will be output during a format.
1281     *
1282     * @param literal the literal to append, not null
1283     * @return this, for chaining, not null
1284     */
1285     public DateTimeFormatterBuilder appendLiteral(char literal) {
1286         appendInternal(new CharLiteralPrinterParser(literal));
1287         return this;
1288     }
1289
1290     /**
1291     * Appends a string literal to the formatter.
1292     * <p>
1293     * This string will be output during a format.
1294     * <p>
1295     * If the literal is empty, nothing is added to the formatter.
1296     *
1297     * @param literal the literal to append, not null
1298     * @return this, for chaining, not null
1299     */
1300     public DateTimeFormatterBuilder appendLiteral(String literal) {
1301         Objects.requireNonNull(literal, "literal");
1302         if (literal.length() > 0) {
1303             if (literal.length() == 1) {
1304                 appendInternal(new CharLiteralPrinterParser(literal.charAt(0)));

```

```

1305         } else {
1306             appendInternal(new StringLiteralPrinterParser(literal));
1307         }
1308     }
1309     return this;
1310 }
1311
1312 //-----
1313 /**
1314  * Appends all the elements of a formatter to the builder.
1315  * <p>
1316  * This method has the same effect as appending each of the constituent
1317  * parts of the formatter directly to this builder.
1318  *
1319  * @param formatter the formatter to add, not null
1320  * @return this, for chaining, not null
1321  */
1322 public DateTimeFormatterBuilder append(DateTimeFormatter formatter) {
1323     Objects.requireNonNull(formatter, "formatter");
1324     appendInternal(formatter.toPrinterParser(false));
1325     return this;
1326 }
1327
1328 /**
1329  * Appends a formatter to the builder which will optionally format/parse.
1330  * <p>
1331  * This method has the same effect as appending each of the constituent
1332  * parts directly to this builder surrounded by an {@link #optionalStart()} and
1333  * {@link #optionalEnd()}.
1334  * <p>
1335  * The formatter will format if data is available for all the fields contained within it.
1336  * The formatter will parse if the string matches, otherwise no error is returned.
1337  *
1338  * @param formatter the formatter to add, not null
1339  * @return this, for chaining, not null
1340  */
1341 public DateTimeFormatterBuilder appendOptional(DateTimeFormatter formatter) {
1342     Objects.requireNonNull(formatter, "formatter");
1343     appendInternal(formatter.toPrinterParser(true));
1344     return this;
1345 }
1346
1347 //-----
1348 /**
1349  * Appends the elements defined by the specified pattern to the builder.
1350  * <p>
1351  * All letters 'A' to 'Z' and 'a' to 'z' are reserved as pattern letters.
1352  * The characters '#', '{' and '}' are reserved for future use.
1353  * The characters '[' and ']' indicate optional patterns.
1354  * The following pattern letters are defined:
1355  * <pre>
1356  * Symbol   Meaning                                Presentation   Examples
1357  * -----

```

```

1358 * G      era      text      AD; Anno Domini; A
1359 * u      year     year      2004; 04
1360 * y      year-of-era year     2004; 04
1361 * D      day-of-year number    189
1362 * M/L    month-of-year number/text 7; 07; Jul; July; J
1363 * d      day-of-month number    10
1364 *
1365 * Q/q    quarter-of-year number/text 3; 03; Q3; 3rd quarter
1366 * Y      week-based-year year     1996; 96
1367 * w      week-of-week-based-year number 27
1368 * W      week-of-month number     4
1369 * E      day-of-week text        Tue; Tuesday; T
1370 * e/c    localized day-of-week number/text 2; 02; Tue; Tuesday; T
1371 * F      week-of-month number     3
1372 *
1373 * a      am-pm-of-day text        PM
1374 * h      clock-hour-of-am-pm (1-12) number 12
1375 * K      hour-of-am-pm (0-11) number 0
1376 * k      clock-hour-of-am-pm (1-24) number 0
1377 *
1378 * H      hour-of-day (0-23) number 0
1379 * m      minute-of-hour number     30
1380 * s      second-of-minute number    55
1381 * S      fraction-of-second fraction 978
1382 * A      milli-of-day number       1234
1383 * n      nano-of-second number     987654321
1384 * N      nano-of-day number       1234000000
1385 *
1386 * V      time-zone ID zone-id     America/Los_Angeles; Z; -08:30
1387 * z      time-zone name zone-name Pacific Standard Time; PST
1388 * O      localized zone-offset offset-0 GMT+8; GMT+08:00; UTC-08:00;
1389 * X      zone-offset 'Z' for zero offset-X Z; -08; -0830; -08:30; -083015;
1390 * x      zone-offset offset-x     +0000; -08; -0830; -08:30; -083015;
1391 * Z      zone-offset offset-Z     +0000; -0800; -08:00;
1392 *
1393 * p      pad next   pad modifier 1
1394 *
1395 * '      escape for text delimiter
1396 * ''     single quote literal    '
1397 * [      optional section start
1398 * ]      optional section end
1399 * #      reserved for future use
1400 * {      reserved for future use
1401 * }      reserved for future use
1402 * </pre>
1403 * <p>
1404 * The count of pattern letters determine the format.
1405 * See <a href="DateTimeFormatter.html#patterns">DateTimeFormatter</a> for a user-focused
1406 * description of the patterns.
1407 * The following tables define how the pattern letters map to the builder.
1408 * <p>

```

```

1408 * <b>Date fields</b>: Pattern letters to output a date.
1409 * <pre>
1410 *   Pattern   Count   Equivalent builder methods
1411 *   -----   -
1412 *   G         1       appendText(ChronoField.ERA, TextStyle.SHORT)
1413 *   GG        2       appendText(ChronoField.ERA, TextStyle.SHORT)
1414 *   GGG       3       appendText(ChronoField.ERA, TextStyle.SHORT)
1415 *   GGGG      4       appendText(ChronoField.ERA, TextStyle.FULL)
1416 *   GGGGG     5       appendText(ChronoField.ERA, TextStyle.NARROW)
1417 *
1418 *   u         1       appendValue(ChronoField.YEAR, 1, 19, SignStyle.NORMAL);
1419 *   uu        2       appendValueReduced(ChronoField.YEAR, 2, 2000);
1420 *   uuu       3       appendValue(ChronoField.YEAR, 3, 19, SignStyle.NORMAL);
1421 *   u..u     4..n     appendValue(ChronoField.YEAR, n, 19, SignStyle.EXCEEDS_PAD);
1422 *   y         1       appendValue(ChronoField.YEAR_OF_ERA, 1, 19, SignStyle.NORMAL);
1423 *   yy        2       appendValueReduced(ChronoField.YEAR_OF_ERA, 2, 2000);
1424 *   yyy       3       appendValue(ChronoField.YEAR_OF_ERA, 3, 19, SignStyle.NORMAL);
1425 *   y..y     4..n     appendValue(ChronoField.YEAR_OF_ERA, n, 19, SignStyle.EXCEEDS_PAD);
1426 *   Y         1       append special localized WeekFields element for numeric week-based-year
1427 *   YY        2       append special localized WeekFields element for reduced numeric week-based
                        -year 2 digits;
1428 *   YYY       3       append special localized WeekFields element for numeric week-based-year
                        (3, 19, SignStyle.NORMAL);
1429 *   Y..Y     4..n     append special localized WeekFields element for numeric week-based-year (n
                        , 19, SignStyle.EXCEEDS_PAD);
1430 *
1431 *   Q         1       appendValue(IsoFields.QUARTER_OF_YEAR);
1432 *   QQ        2       appendValue(IsoFields.QUARTER_OF_YEAR, 2);
1433 *   QQQ       3       appendText(IsoFields.QUARTER_OF_YEAR, TextStyle.SHORT)
1434 *   QQQQ      4       appendText(IsoFields.QUARTER_OF_YEAR, TextStyle.FULL)
1435 *   QQQQQ     5       appendText(IsoFields.QUARTER_OF_YEAR, TextStyle.NARROW)
1436 *   q         1       appendValue(IsoFields.QUARTER_OF_YEAR);
1437 *   qq        2       appendValue(IsoFields.QUARTER_OF_YEAR, 2);
1438 *   qqq       3       appendText(IsoFields.QUARTER_OF_YEAR, TextStyle.SHORT_STANDALONE)
1439 *   qqqq      4       appendText(IsoFields.QUARTER_OF_YEAR, TextStyle.FULL_STANDALONE)
1440 *   qqqqq     5       appendText(IsoFields.QUARTER_OF_YEAR, TextStyle.NARROW_STANDALONE)
1441 *
1442 *   M         1       appendValue(ChronoField.MONTH_OF_YEAR);
1443 *   MM        2       appendValue(ChronoField.MONTH_OF_YEAR, 2);
1444 *   MMM       3       appendText(ChronoField.MONTH_OF_YEAR, TextStyle.SHORT)
1445 *   MMMM      4       appendText(ChronoField.MONTH_OF_YEAR, TextStyle.FULL)
1446 *   MMMMM     5       appendText(ChronoField.MONTH_OF_YEAR, TextStyle.NARROW)
1447 *   L         1       appendValue(ChronoField.MONTH_OF_YEAR);
1448 *   LL        2       appendValue(ChronoField.MONTH_OF_YEAR, 2);
1449 *   LLL       3       appendText(ChronoField.MONTH_OF_YEAR, TextStyle.SHORT_STANDALONE)
1450 *   LLLL      4       appendText(ChronoField.MONTH_OF_YEAR, TextStyle.FULL_STANDALONE)
1451 *   LLLLL     5       appendText(ChronoField.MONTH_OF_YEAR, TextStyle.NARROW_STANDALONE)
1452 *
1453 *   w         1       append special localized WeekFields element for numeric week-of-year
1454 *   ww        2       append special localized WeekFields element for numeric week-of-year, zero
                        -padded
1455 *   W         1       append special localized WeekFields element for numeric week-of-month
1456 *   d         1       appendValue(ChronoField.DAY_OF_MONTH)

```

```

1457 *   dd      2      appendValue(ChronoField.DAY_OF_MONTH, 2)
1458 *   D        1      appendValue(ChronoField.DAY_OF_YEAR)
1459 *   DD       2      appendValue(ChronoField.DAY_OF_YEAR, 2)
1460 *   DDD      3      appendValue(ChronoField.DAY_OF_YEAR, 3)
1461 *   F        1      appendValue(ChronoField.ALIGNED_DAY_OF_WEEK_IN_MONTH)
1462 *   E        1      appendText(ChronoField.DAY_OF_WEEK, TextStyle.SHORT)
1463 *   EE       2      appendText(ChronoField.DAY_OF_WEEK, TextStyle.SHORT)
1464 *   EEE      3      appendText(ChronoField.DAY_OF_WEEK, TextStyle.SHORT)
1465 *   EEEE     4      appendText(ChronoField.DAY_OF_WEEK, TextStyle.FULL)
1466 *   EEEEE    5      appendText(ChronoField.DAY_OF_WEEK, TextStyle.NARROW)
1467 *   e        1      append special localized WeekFields element for numeric day-of-week
1468 *   ee       2      append special localized WeekFields element for numeric day-of-week, zero-
                        padded
1469 *   eee      3      appendText(ChronoField.DAY_OF_WEEK, TextStyle.SHORT)
1470 *   eeee     4      appendText(ChronoField.DAY_OF_WEEK, TextStyle.FULL)
1471 *   eeeee    5      appendText(ChronoField.DAY_OF_WEEK, TextStyle.NARROW)
1472 *   c        1      append special localized WeekFields element for numeric day-of-week
1473 *   ccc      3      appendText(ChronoField.DAY_OF_WEEK, TextStyle.SHORT_STANDALONE)
1474 *   cccc     4      appendText(ChronoField.DAY_OF_WEEK, TextStyle.FULL_STANDALONE)
1475 *   ccccc    5      appendText(ChronoField.DAY_OF_WEEK, TextStyle.NARROW_STANDALONE)
1476 * </pre>
1477 * <p>
1478 * <b>Time fields</b>: Pattern letters to output a time.
1479 * <pre>
1480 * Pattern Count Equivalent builder methods
1481 * -----
1482 *   a        1      appendText(ChronoField.AMPM_OF_DAY, TextStyle.SHORT)
1483 *   h        1      appendValue(ChronoField.CLOCK_HOUR_OF_AMPM)
1484 *   hh       2      appendValue(ChronoField.CLOCK_HOUR_OF_AMPM, 2)
1485 *   H        1      appendValue(ChronoField.HOUR_OF_DAY)
1486 *   HH       2      appendValue(ChronoField.HOUR_OF_DAY, 2)
1487 *   k        1      appendValue(ChronoField.CLOCK_HOUR_OF_DAY)
1488 *   kk       2      appendValue(ChronoField.CLOCK_HOUR_OF_DAY, 2)
1489 *   K        1      appendValue(ChronoField.HOUR_OF_AMPM)
1490 *   KK       2      appendValue(ChronoField.HOUR_OF_AMPM, 2)
1491 *   m        1      appendValue(ChronoField.MINUTE_OF_HOUR)
1492 *   mm       2      appendValue(ChronoField.MINUTE_OF_HOUR, 2)
1493 *   s        1      appendValue(ChronoField.SECOND_OF_MINUTE)
1494 *   ss       2      appendValue(ChronoField.SECOND_OF_MINUTE, 2)
1495 *
1496 *   S..S     1..n    appendFraction(ChronoField.NANO_OF_SECOND, n, n, false)
1497 *   A        1      appendValue(ChronoField.MILLI_OF_DAY)
1498 *   A..A     2..n    appendValue(ChronoField.MILLI_OF_DAY, n)
1499 *   n        1      appendValue(ChronoField.NANO_OF_SECOND)
1500 *   n..n     2..n    appendValue(ChronoField.NANO_OF_SECOND, n)
1501 *   N        1      appendValue(ChronoField.NANO_OF_DAY)
1502 *   N..N     2..n    appendValue(ChronoField.NANO_OF_DAY, n)
1503 * </pre>
1504 * <p>
1505 * <b>Zone ID</b>: Pattern letters to output {@code ZoneId}.
1506 * <pre>
1507 * Pattern Count Equivalent builder methods
1508 * -----

```

```

1509 *   VV      2      appendZoneId()
1510 *   z        1      appendZoneText(TextStyle.SHORT)
1511 *   zz       2      appendZoneText(TextStyle.SHORT)
1512 *   zzz      3      appendZoneText(TextStyle.SHORT)
1513 *   zzzz     4      appendZoneText(TextStyle.FULL)
1514 * </pre>
1515 * <p>
1516 * <b>Zone offset</b>: Pattern letters to output {@code ZoneOffset}.
1517 * <pre>
1518 *   Pattern  Count  Equivalent builder methods
1519 *   -----  -
1520 *   O         1      appendLocalizedOffsetPrefixed(TextStyle.SHORT);
1521 *   OOOO      4      appendLocalizedOffsetPrefixed(TextStyle.FULL);
1522 *   X         1      appendOffset("+HHmm","Z")
1523 *   XX        2      appendOffset("+HHMM","Z")
1524 *   XXX       3      appendOffset("+HH:MM","Z")
1525 *   XXXX      4      appendOffset("+HHMMss","Z")
1526 *   XXXXX     5      appendOffset("+HH:MM:ss","Z")
1527 *   x         1      appendOffset("+HHmm","+00")
1528 *   xx        2      appendOffset("+HHMM","+0000")
1529 *   xxx       3      appendOffset("+HH:MM","+00:00")
1530 *   xxxx      4      appendOffset("+HHMMss","+0000")
1531 *   xxxxx     5      appendOffset("+HH:MM:ss","+00:00")
1532 *   Z         1      appendOffset("+HHMM","+0000")
1533 *   ZZ        2      appendOffset("+HHMM","+0000")
1534 *   ZZZ       3      appendOffset("+HHMM","+0000")
1535 *   ZZZZ      4      appendLocalizedOffset(TextStyle.FULL);
1536 *   ZZZZZ     5      appendOffset("+HH:MM:ss","Z")
1537 * </pre>
1538 * <p>
1539 * <b>Modifiers</b>: Pattern letters that modify the rest of the pattern:
1540 * <pre>
1541 *   Pattern  Count  Equivalent builder methods
1542 *   -----  -
1543 *   [         1      optionalStart()
1544 *   ]         1      optionalEnd()
1545 *   p..p     1..n    padNext(n)
1546 * </pre>
1547 * <p>
1548 * Any sequence of letters not specified above, unrecognized letter or
1549 * reserved character will throw an exception.
1550 * Future versions may add to the set of patterns.
1551 * It is recommended to use single quotes around all characters that you want
1552 * to output directly to ensure that future changes do not break your application.
1553 * <p>
1554 * Note that the pattern string is similar, but not identical, to
1555 * {@link java.text.SimpleDateFormat SimpleDateFormat}.
1556 * The pattern string is also similar, but not identical, to that defined by the
1557 * Unicode Common Locale Data Repository (CLDR/LDML).
1558 * Pattern letters 'X' and 'u' are aligned with Unicode CLDR/LDML.
1559 * By contrast, {@code SimpleDateFormat} uses 'u' for the numeric day of week.
1560 * Pattern letters 'y' and 'Y' parse years of two digits and more than 4 digits differently.
1561 * Pattern letters 'n', 'A', 'N', and 'p' are added.

```

```

1562     * Number types will reject large numbers.
1563     *
1564     * @param pattern the pattern to add, not null
1565     * @return this, for chaining, not null
1566     * @throws IllegalArgumentException if the pattern is invalid
1567     */
1568     public DateTimeFormatterBuilder appendPattern(String pattern) {
1569         Objects.requireNonNull(pattern, "pattern");
1570         parsePattern(pattern);
1571         return this;
1572     }
1573
1574     private void parsePattern(String pattern) {
1575         for (int pos = 0; pos < pattern.length(); pos++) {
1576             char cur = pattern.charAt(pos);
1577             if ((cur >= 'A' && cur <= 'Z') || (cur >= 'a' && cur <= 'z')) {
1578                 int start = pos++;
1579                 for ( ; pos < pattern.length() && pattern.charAt(pos) == cur; pos++); // short
                    loop
1580                 int count = pos - start;
1581                 // padding
1582                 if (cur == 'p') {
1583                     int pad = 0;
1584                     if (pos < pattern.length()) {
1585                         cur = pattern.charAt(pos);
1586                         if ((cur >= 'A' && cur <= 'Z') || (cur >= 'a' && cur <= 'z')) {
1587                             pad = count;
1588                             start = pos++;
1589                             for ( ; pos < pattern.length() && pattern.charAt(pos) == cur; pos++);
                                    // short loop
1590                             count = pos - start;
1591                         }
1592                     }
1593                     if (pad == 0) {
1594                         throw new IllegalArgumentException(
1595                             "Pad letter 'p' must be followed by valid pad pattern: " + pattern)
1596                             ;
1597                     }
1598                     padNext(pad); // pad and continue parsing
1599                 }
1600                 // main rules
1601                 TemporalField field = FIELD_MAP.get(cur);
1602                 if (field != null) {
1603                     parseField(cur, count, field);
1604                 } else if (cur == 'z') {
1605                     if (count > 4) {
1606                         throw new IllegalArgumentException("Too many pattern letters: " + cur);
1607                     } else if (count == 4) {
1608                         appendZoneText(TextStyle.FULL);
1609                     } else {
1610                         appendZoneText(TextStyle.SHORT);
1611                     }
1612                 } else if (cur == 'V') {

```



```

1612         if (count != 2) {
1613             throw new IllegalArgumentException("Pattern letter count must be 2: " + cur
1614                 );
1615         }
1616         appendZoneId();
1617     } else if (cur == 'Z') {
1618         if (count < 4) {
1619             appendOffset("+HHMM", "+0000");
1620         } else if (count == 4) {
1621             appendLocalizedOffset(TextStyle.FULL);
1622         } else if (count == 5) {
1623             appendOffset("+HH:MM:ss", "Z");
1624         } else {
1625             throw new IllegalArgumentException("Too many pattern letters: " + cur);
1626         }
1627     } else if (cur == 'O') {
1628         if (count == 1) {
1629             appendLocalizedOffset(TextStyle.SHORT);
1630         } else if (count == 4) {
1631             appendLocalizedOffset(TextStyle.FULL);
1632         } else {
1633             throw new IllegalArgumentException("Pattern letter count must be 1 or 4: "
1634                 + cur);
1635         }
1636     } else if (cur == 'X') {
1637         if (count > 5) {
1638             throw new IllegalArgumentException("Too many pattern letters: " + cur);
1639         }
1640         appendOffset(OffsetIdPrinterParser.PATTERNS[count + (count == 1 ? 0 : 1)], "Z")
1641             ;
1642     } else if (cur == 'x') {
1643         if (count > 5) {
1644             throw new IllegalArgumentException("Too many pattern letters: " + cur);
1645         }
1646         String zero = (count == 1 ? "+00" : (count % 2 == 0 ? "+0000" : "+00:00"));
1647         appendOffset(OffsetIdPrinterParser.PATTERNS[count + (count == 1 ? 0 : 1)], zero
1648             );
1649     } else if (cur == 'W') {
1650         // Fields defined by Locale
1651         if (count > 1) {
1652             throw new IllegalArgumentException("Too many pattern letters: " + cur);
1653         }
1654         appendInternal(new WeekBasedFieldPrinterParser(cur, count));
1655     } else if (cur == 'w') {
1656         // Fields defined by Locale
1657         if (count > 2) {
1658             throw new IllegalArgumentException("Too many pattern letters: " + cur);
1659         }
1660         appendInternal(new WeekBasedFieldPrinterParser(cur, count));
1661     } else if (cur == 'Y') {
1662         // Fields defined by Locale
1663         appendInternal(new WeekBasedFieldPrinterParser(cur, count));
1664     } else {

```



```

1661         throw new IllegalArgumentException("Unknown pattern letter: " + cur);
1662     }
1663     pos--;
1664
1665     } else if (cur == '\\') {
1666         // parse literals
1667         int start = pos++;
1668         for ( ; pos < pattern.length(); pos++) {
1669             if (pattern.charAt(pos) == '\\') {
1670                 if (pos + 1 < pattern.length() && pattern.charAt(pos + 1) == '\\') {
1671                     pos++;
1672                 } else {
1673                     break; // end of literal
1674                 }
1675             }
1676         }
1677         if (pos >= pattern.length()) {
1678             throw new IllegalArgumentException("Pattern ends with an incomplete string
1679                 literal: " + pattern);
1680         }
1681         String str = pattern.substring(start + 1, pos);
1682         if (str.length() == 0) {
1683             appendLiteral('\\');
1684         } else {
1685             appendLiteral(str.replace("'", ""));
1686         }
1687     } else if (cur == '[') {
1688         optionalStart();
1689
1690     } else if (cur == ']') {
1691         if (active.parent == null) {
1692             throw new IllegalArgumentException("Pattern invalid as it contains ] without
1693                 previous [");
1694         }
1695         optionalEnd();
1696
1697     } else if (cur == '{' || cur == '}' || cur == '#') {
1698         throw new IllegalArgumentException("Pattern includes reserved character: '" + cur +
1699             "'");
1700     } else {
1701         appendLiteral(cur);
1702     }
1703 }
1704
1705 @SuppressWarnings("fallthrough")
1706 private void parseField(char cur, int count, TemporalField field) {
1707     boolean standalone = false;
1708     switch (cur) {
1709         case 'u':
1710         case 'y':
1711             if (count == 2) {

```

```
1711         appendValueReduced(field, 2, 2, ReducedPrinterParser.BASE_DATE);
1712     } else if (count < 4) {
1713         appendValue(field, count, 19, SignStyle.NORMAL);
1714     } else {
1715         appendValue(field, count, 19, SignStyle.EXCEEDS_PAD);
1716     }
1717     break;
1718 case 'c':
1719     if (count == 2) {
1720         throw new IllegalArgumentException("Invalid pattern \"cc\"");
1721     }
1722     /*fallthrough*/
1723 case 'L':
1724 case 'q':
1725     standalone = true;
1726     /*fallthrough*/
1727 case 'M':
1728 case 'Q':
1729 case 'E':
1730 case 'e':
1731     switch (count) {
1732         case 1:
1733         case 2:
1734             if (cur == 'c' || cur == 'e') {
1735                 appendInternal(new WeekBasedFieldPrinterParser(cur, count));
1736             } else if (cur == 'E') {
1737                 appendText(field, TextStyle.SHORT);
1738             } else {
1739                 if (count == 1) {
1740                     appendValue(field);
1741                 } else {
1742                     appendValue(field, 2);
1743                 }
1744             }
1745             break;
1746         case 3:
1747             appendText(field, standalone ? TextStyle.SHORT_STANDALONE : TextStyle.SHORT);
1748             break;
1749         case 4:
1750             appendText(field, standalone ? TextStyle.FULL_STANDALONE : TextStyle.FULL);
1751             break;
1752         case 5:
1753             appendText(field, standalone ? TextStyle.NARROW_STANDALONE : TextStyle.NARROW);
1754             break;
1755         default:
1756             throw new IllegalArgumentException("Too many pattern letters: " + cur);
1757     }
1758     break;
1759 case 'a':
1760     if (count == 1) {
1761         appendText(field, TextStyle.SHORT);
```

```
1762         } else {
1763             throw new IllegalArgumentException("Too many pattern letters: " + cur);
1764         }
1765         break;
1766     case 'G':
1767         switch (count) {
1768             case 1:
1769             case 2:
1770             case 3:
1771                 appendText(field, TextStyle.SHORT);
1772                 break;
1773             case 4:
1774                 appendText(field, TextStyle.FULL);
1775                 break;
1776             case 5:
1777                 appendText(field, TextStyle.NARROW);
1778                 break;
1779             default:
1780                 throw new IllegalArgumentException("Too many pattern letters: " + cur);
1781         }
1782         break;
1783     case 'S':
1784         appendFraction(NANO_OF_SECOND, count, count, false);
1785         break;
1786     case 'F':
1787         if (count == 1) {
1788             appendValue(field);
1789         } else {
1790             throw new IllegalArgumentException("Too many pattern letters: " + cur);
1791         }
1792         break;
1793     case 'd':
1794     case 'h':
1795     case 'H':
1796     case 'k':
1797     case 'K':
1798     case 'm':
1799     case 's':
1800         if (count == 1) {
1801             appendValue(field);
1802         } else if (count == 2) {
1803             appendValue(field, count);
1804         } else {
1805             throw new IllegalArgumentException("Too many pattern letters: " + cur);
1806         }
1807         break;
1808     case 'D':
1809         if (count == 1) {
1810             appendValue(field);
1811         } else if (count <= 3) {
1812             appendValue(field, count);
1813         } else {
1814             throw new IllegalArgumentException("Too many pattern letters: " + cur);
```

```

1815     }
1816     break;
1817     default:
1818         if (count == 1) {
1819             appendValue(field);
1820         } else {
1821             appendValue(field, count);
1822         }
1823         break;
1824     }
1825 }
1826
1827 /** Map of letters to fields. */
1828 private static final Map<Character, TemporalField> FIELD_MAP = new HashMap<>();
1829 static {
1830     // SDF = SimpleDateFormat
1831     FIELD_MAP.put('G', ChronoField.ERA); // SDF, LDML (different to both
1832         for 1/2 chars)
1833     FIELD_MAP.put('y', ChronoField.YEAR_OF_ERA); // SDF, LDML
1834     FIELD_MAP.put('u', ChronoField.YEAR); // LDML (different in SDF)
1835     FIELD_MAP.put('Q', IsoFields.QUARTER_OF_YEAR); // LDML (removed quarter from
1836         310)
1837     FIELD_MAP.put('q', IsoFields.QUARTER_OF_YEAR); // LDML (stand-alone)
1838     FIELD_MAP.put('M', ChronoField.MONTH_OF_YEAR); // SDF, LDML
1839     FIELD_MAP.put('L', ChronoField.MONTH_OF_YEAR); // SDF, LDML (stand-alone)
1840     FIELD_MAP.put('D', ChronoField.DAY_OF_YEAR); // SDF, LDML
1841     FIELD_MAP.put('d', ChronoField.DAY_OF_MONTH); // SDF, LDML
1842     FIELD_MAP.put('F', ChronoField.ALIGNED_DAY_OF_WEEK_IN_MONTH); // SDF, LDML
1843     FIELD_MAP.put('E', ChronoField.DAY_OF_WEEK); // SDF, LDML (different to both
1844         for 1/2 chars)
1845     FIELD_MAP.put('c', ChronoField.DAY_OF_WEEK); // LDML (stand-alone)
1846     FIELD_MAP.put('e', ChronoField.DAY_OF_WEEK); // LDML (needs localized week
1847         number)
1848     FIELD_MAP.put('a', ChronoField.AMPM_OF_DAY); // SDF, LDML
1849     FIELD_MAP.put('H', ChronoField.HOUR_OF_DAY); // SDF, LDML
1850     FIELD_MAP.put('k', ChronoField.CLOCK_HOUR_OF_DAY); // SDF, LDML
1851     FIELD_MAP.put('K', ChronoField.HOUR_OF_AMPM); // SDF, LDML
1852     FIELD_MAP.put('h', ChronoField.CLOCK_HOUR_OF_AMPM); // SDF, LDML
1853     FIELD_MAP.put('m', ChronoField.MINUTE_OF_HOUR); // SDF, LDML
1854     FIELD_MAP.put('s', ChronoField.SECOND_OF_MINUTE); // SDF, LDML
1855     FIELD_MAP.put('S', ChronoField.NANO_OF_SECOND); // LDML (SDF uses milli-of-
1856         second number)
1857     FIELD_MAP.put('A', ChronoField.MILLI_OF_DAY); // LDML
1858     FIELD_MAP.put('n', ChronoField.NANO_OF_SECOND); // 310 (proposed for LDML)
1859     FIELD_MAP.put('N', ChronoField.NANO_OF_DAY); // 310 (proposed for LDML)
1860     // 310 - z - time-zone names, matches LDML and SimpleDateFormat 1 to 4
1861     // 310 - Z - matches SimpleDateFormat and LDML
1862     // 310 - V - time-zone id, matches LDML
1863     // 310 - p - prefix for padding
1864     // 310 - X - matches LDML, almost matches SDF for 1, exact match 2&3, extended 4&5
1865     // 310 - x - matches LDML
1866     // 310 - w, W, and Y are localized forms matching LDML
1867     // LDML - U - cycle year name, not supported by 310 yet

```

```

1863         // LDML - l - deprecated
1864         // LDML - j - not relevant
1865         // LDML - g - modified-julian-day
1866         // LDML - v,V - extended time-zone names
1867     }
1868
1869     //-----
1870     /**
1871      * Causes the next added printer/parser to pad to a fixed width using a space.
1872      * <p>
1873      * This padding will pad to a fixed width using spaces.
1874      * <p>
1875      * During formatting, the decorated element will be output and then padded
1876      * to the specified width. An exception will be thrown during formatting if
1877      * the pad width is exceeded.
1878      * <p>
1879      * During parsing, the padding and decorated element are parsed.
1880      * If parsing is lenient, then the pad width is treated as a maximum.
1881      * The padding is parsed greedily. Thus, if the decorated element starts with
1882      * the pad character, it will not be parsed.
1883      *
1884      * @param padWidth the pad width, 1 or greater
1885      * @return this, for chaining, not null
1886      * @throws IllegalArgumentException if pad width is too small
1887      */
1888     public DateTimeFormatterBuilder padNext(int padWidth) {
1889         return padNext(padWidth, ' ');
1890     }
1891
1892     /**
1893      * Causes the next added printer/parser to pad to a fixed width.
1894      * <p>
1895      * This padding is intended for padding other than zero-padding.
1896      * Zero-padding should be achieved using the appendValue methods.
1897      * <p>
1898      * During formatting, the decorated element will be output and then padded
1899      * to the specified width. An exception will be thrown during formatting if
1900      * the pad width is exceeded.
1901      * <p>
1902      * During parsing, the padding and decorated element are parsed.
1903      * If parsing is lenient, then the pad width is treated as a maximum.
1904      * If parsing is case insensitive, then the pad character is matched ignoring case.
1905      * The padding is parsed greedily. Thus, if the decorated element starts with
1906      * the pad character, it will not be parsed.
1907      *
1908      * @param padWidth the pad width, 1 or greater
1909      * @param padChar the pad character
1910      * @return this, for chaining, not null
1911      * @throws IllegalArgumentException if pad width is too small
1912      */
1913     public DateTimeFormatterBuilder padNext(int padWidth, char padChar) {
1914         if (padWidth < 1) {
1915             throw new IllegalArgumentException("The pad width must be at least one but was " +

```

```

        padWidth);
1916     }
1917     active.padNextWidth = padWidth;
1918     active.padNextChar = padChar;
1919     active.valueParserIndex = -1;
1920     return this;
1921 }
1922
1923 //-----
1924 /**
1925  * Mark the start of an optional section.
1926  * <p>
1927  * The output of formatting can include optional sections, which may be nested.
1928  * An optional section is started by calling this method and ended by calling
1929  * {@link #optionalEnd()} or by ending the build process.
1930  * <p>
1931  * All elements in the optional section are treated as optional.
1932  * During formatting, the section is only output if data is available in the
1933  * {@code TemporalAccessor} for all the elements in the section.
1934  * During parsing, the whole section may be missing from the parsed string.
1935  * <p>
1936  * For example, consider a builder setup as
1937  * {@code builder.appendValue(HOUR_OF_DAY,2).optionalStart().appendValue(MINUTE_OF_HOUR,2)}.
1938  * The optional section ends automatically at the end of the builder.
1939  * During formatting, the minute will only be output if its value can be obtained from the date
1940  * -time.
1941  * During parsing, the input will be successfully parsed whether the minute is present or not.
1942  *
1943  * @return this, for chaining, not null
1944  */
1945 public DateTimeFormatterBuilder optionalStart() {
1946     active.valueParserIndex = -1;
1947     active = new DateTimeFormatterBuilder(active, true);
1948     return this;
1949 }
1950
1951 /**
1952  * Ends an optional section.
1953  * <p>
1954  * The output of formatting can include optional sections, which may be nested.
1955  * An optional section is started by calling {@link #optionalStart()} and ended
1956  * using this method (or at the end of the builder).
1957  * <p>
1958  * Calling this method without having previously called {@code optionalStart}
1959  * will throw an exception.
1960  * Calling this method immediately after calling {@code optionalStart} has no effect
1961  * on the formatter other than ending the (empty) optional section.
1962  * <p>
1963  * All elements in the optional section are treated as optional.
1964  * During formatting, the section is only output if data is available in the
1965  * {@code TemporalAccessor} for all the elements in the section.
1966  * During parsing, the whole section may be missing from the parsed string.
1967  * <p>

```

```

1967 * For example, consider a builder setup as
1968 * {@code builder.appendValue(HOUR_OF_DAY,2).optionalStart().appendValue(MINUTE_OF_HOUR,2).
    optionalEnd()}.
1969 * During formatting, the minute will only be output if its value can be obtained from the date
    -time.
1970 * During parsing, the input will be successfully parsed whether the minute is present or not.
1971 *
1972 * @return this, for chaining, not null
1973 * @throws IllegalStateException if there was no previous call to {@code optionalStart}
1974 */
1975 public DateTimeFormatterBuilder optionalEnd() {
1976     if (active.parent == null) {
1977         throw new IllegalStateException("Cannot call optionalEnd() as there was no previous
            call to optionalStart()");
1978     }
1979     if (active.printerParsers.size() > 0) {
1980         CompositePrinterParser cpp = new CompositePrinterParser(active.printerParsers, active.
            optional);
1981         active = active.parent;
1982         appendInternal(cpp);
1983     } else {
1984         active = active.parent;
1985     }
1986     return this;
1987 }
1988
1989 //-----
1990 /**
1991  * Appends a printer and/or parser to the internal list handling padding.
1992  *
1993  * @param pp the printer-parser to add, not null
1994  * @return the index into the active parsers list
1995  */
1996 private int appendInternal(DateTimePrinterParser pp) {
1997     Objects.requireNonNull(pp, "pp");
1998     if (active.padNextWidth > 0) {
1999         if (pp != null) {
2000             pp = new PadPrinterParserDecorator(pp, active.padNextWidth, active.padNextChar);
2001         }
2002         active.padNextWidth = 0;
2003         active.padNextChar = 0;
2004     }
2005     active.printerParsers.add(pp);
2006     active.valueParserIndex = -1;
2007     return active.printerParsers.size() - 1;
2008 }
2009
2010 //-----
2011 /**
2012  * Completes this builder by creating the {@code DateTimeFormatter}
2013  * using the default locale.
2014  * <p>
2015  * This will create a formatter with the {@linkplain Locale#getDefault(Locale.Category) default

```

```

    FORMAT locale}.
2016     * Numbers will be printed and parsed using the standard DecimalStyle.
2017     * The resolver style will be {@link ResolverStyle#SMART SMART}.
2018     * <p>
2019     * Calling this method will end any open optional sections by repeatedly
2020     * calling {@link #optionalEnd()} before creating the formatter.
2021     * <p>
2022     * This builder can still be used after creating the formatter if desired,
2023     * although the state may have been changed by calls to {@code optionalEnd}.
2024     *
2025     * @return the created formatter, not null
2026     */
2027     public DateTimeFormatter toFormatter() {
2028         return toFormatter(Locale.getDefault(Locale.Category.FORMAT));
2029     }
2030
2031     /**
2032     * Completes this builder by creating the {@code DateTimeFormatter}
2033     * using the specified locale.
2034     * <p>
2035     * This will create a formatter with the specified locale.
2036     * Numbers will be printed and parsed using the standard DecimalStyle.
2037     * The resolver style will be {@link ResolverStyle#SMART SMART}.
2038     * <p>
2039     * Calling this method will end any open optional sections by repeatedly
2040     * calling {@link #optionalEnd()} before creating the formatter.
2041     * <p>
2042     * This builder can still be used after creating the formatter if desired,
2043     * although the state may have been changed by calls to {@code optionalEnd}.
2044     *
2045     * @param locale the locale to use for formatting, not null
2046     * @return the created formatter, not null
2047     */
2048     public DateTimeFormatter toFormatter(Locale locale) {
2049         return toFormatter(locale, ResolverStyle.SMART, null);
2050     }
2051
2052     /**
2053     * Completes this builder by creating the formatter.
2054     * This uses the default locale.
2055     *
2056     * @param resolverStyle the resolver style to use, not null
2057     * @return the created formatter, not null
2058     */
2059     DateTimeFormatter toFormatter(ResolverStyle resolverStyle, Chronology chrono) {
2060         return toFormatter(Locale.getDefault(Locale.Category.FORMAT), resolverStyle, chrono);
2061     }
2062
2063     /**
2064     * Completes this builder by creating the formatter.
2065     *
2066     * @param locale the locale to use for formatting, not null
2067     * @param chrono the chronology to use, may be null

```



```

2068     * @return the created formatter, not null
2069     */
2070     private DateTimeFormatter toFormatter(Locale locale, ResolverStyle resolverStyle, Chronology
        chrono) {
2071         Objects.requireNonNull(locale, "locale");
2072         while (active.parent != null) {
2073             optionalEnd();
2074         }
2075         CompositePrinterParser pp = new CompositePrinterParser(printerParsers, false);
2076         return new DateTimeFormatter(pp, locale, DecimalStyle.STANDARD,
2077             resolverStyle, null, chrono, null);
2078     }
2079
2080     //-----
2081     /**
2082      * Strategy for formatting/parsing date-time information.
2083      * <p>
2084      * The printer may format any part, or the whole, of the input date-time object.
2085      * Typically, a complete format is constructed from a number of smaller
2086      * units, each outputting a single field.
2087      * <p>
2088      * The parser may parse any piece of text from the input, storing the result
2089      * in the context. Typically, each individual parser will just parse one
2090      * field, such as the day-of-month, storing the value in the context.
2091      * Once the parse is complete, the caller will then resolve the parsed values
2092      * to create the desired object, such as a {@code LocalDate}.
2093      * <p>
2094      * The parse position will be updated during the parse. Parsing will start at
2095      * the specified index and the return value specifies the new parse position
2096      * for the next parser. If an error occurs, the returned index will be negative
2097      * and will have the error position encoded using the complement operator.
2098      *
2099      * @implSpec
2100      * This interface must be implemented with care to ensure other classes operate correctly.
2101      * All implementations that can be instantiated must be final, immutable and thread-safe.
2102      * <p>
2103      * The context is not a thread-safe object and a new instance will be created
2104      * for each format that occurs. The context must not be stored in an instance
2105      * variable or shared with any other threads.
2106      */
2107     interface DateTimePrinterParser {
2108
2109         /**
2110          * Prints the date-time object to the buffer.
2111          * <p>
2112          * The context holds information to use during the format.
2113          * It also contains the date-time information to be printed.
2114          * <p>
2115          * The buffer must not be mutated beyond the content controlled by the implementation.
2116          *
2117          * @param context the context to format using, not null
2118          * @param buf the buffer to append to, not null
2119          * @return false if unable to query the value from the date-time, true otherwise

```

```

2120     * @throws DateTimeException if the date-time cannot be printed successfully
2121     */
2122     boolean format(DateTimePrintContext context, StringBuilder buf);
2123
2124     /**
2125     * Parses text into date-time information.
2126     * <p>
2127     * The context holds information to use during the parse.
2128     * It is also used to store the parsed date-time information.
2129     *
2130     * @param context the context to use and parse into, not null
2131     * @param text the input text to parse, not null
2132     * @param position the position to start parsing at, from 0 to the text length
2133     * @return the new parse position, where negative means an error with the
2134     *         error position encoded using the complement ~ operator
2135     * @throws NullPointerException if the context or text is null
2136     * @throws IndexOutOfBoundsException if the position is invalid
2137     */
2138     int parse(DateTimeParseContext context, CharSequence text, int position);
2139 }
2140
2141 //-----
2142 /**
2143  * Composite printer and parser.
2144  */
2145 static final class CompositePrinterParser implements DateTimePrinterParser {
2146     private final DateTimePrinterParser[] printerParsers;
2147     private final boolean optional;
2148
2149     CompositePrinterParser(List<DateTimePrinterParser> printerParsers, boolean optional) {
2150         this(printerParsers.toArray(new DateTimePrinterParser[printerParsers.size()]), optional);
2151     }
2152
2153     CompositePrinterParser(DateTimePrinterParser[] printerParsers, boolean optional) {
2154         this.printerParsers = printerParsers;
2155         this.optional = optional;
2156     }
2157
2158     /**
2159     * Returns a copy of this printer-parser with the optional flag changed.
2160     *
2161     * @param optional the optional flag to set in the copy
2162     * @return the new printer-parser, not null
2163     */
2164     public CompositePrinterParser withOptional(boolean optional) {
2165         if (optional == this.optional) {
2166             return this;
2167         }
2168         return new CompositePrinterParser(printerParsers, optional);
2169     }
2170
2171     @Override

```

```
2172     public boolean format(DateTimePrintContext context, StringBuilder buf) {
2173         int length = buf.length();
2174         if (optional) {
2175             context.startOptional();
2176         }
2177         try {
2178             for (DateTimePrinterParser pp : printerParsers) {
2179                 if (pp.format(context, buf) == false) {
2180                     buf.setLength(length); // reset buffer
2181                     return true;
2182                 }
2183             }
2184         } finally {
2185             if (optional) {
2186                 context.endOptional();
2187             }
2188         }
2189         return true;
2190     }
2191
2192     @Override
2193     public int parse(DateTimeParseContext context, CharSequence text, int position) {
2194         if (optional) {
2195             context.startOptional();
2196             int pos = position;
2197             for (DateTimePrinterParser pp : printerParsers) {
2198                 pos = pp.parse(context, text, pos);
2199                 if (pos < 0) {
2200                     context.endOptional(false);
2201                     return position; // return original position
2202                 }
2203             }
2204             context.endOptional(true);
2205             return pos;
2206         } else {
2207             for (DateTimePrinterParser pp : printerParsers) {
2208                 position = pp.parse(context, text, position);
2209                 if (position < 0) {
2210                     break;
2211                 }
2212             }
2213             return position;
2214         }
2215     }
2216
2217     @Override
2218     public String toString() {
2219         StringBuilder buf = new StringBuilder();
2220         if (printerParsers != null) {
2221             buf.append(optional ? "[" : "(");
2222             for (DateTimePrinterParser pp : printerParsers) {
2223                 buf.append(pp);
2224             }

```

```

2225         buf.append(optional ? "]" : "");
2226     }
2227     return buf.toString();
2228 }
2229 }
2230
2231 //-----
2232 /**
2233  * Pads the output to a fixed width.
2234  */
2235 static final class PadPrinterParserDecorator implements DateTimePrinterParser {
2236     private final DateTimePrinterParser printerParser;
2237     private final int padWidth;
2238     private final char padChar;
2239
2240     /**
2241      * Constructor.
2242      *
2243      * @param printerParser the printer, not null
2244      * @param padWidth the width to pad to, 1 or greater
2245      * @param padChar the pad character
2246      */
2247     PadPrinterParserDecorator(DateTimePrinterParser printerParser, int padWidth, char padChar)
2248     {
2249         // input checked by DateTimeFormatterBuilder
2250         this.printerParser = printerParser;
2251         this.padWidth = padWidth;
2252         this.padChar = padChar;
2253     }
2254
2255     @Override
2256     public boolean format(DateTimePrintContext context, StringBuilder buf) {
2257         int preLen = buf.length();
2258         if (printerParser.format(context, buf) == false) {
2259             return false;
2260         }
2261         int len = buf.length() - preLen;
2262         if (len > padWidth) {
2263             throw new DateTimeException(
2264                 "Cannot print as output of " + len + " characters exceeds pad width of " +
2265                 padWidth);
2266         }
2267         for (int i = 0; i < padWidth - len; i++) {
2268             buf.insert(preLen, padChar);
2269         }
2270         return true;
2271     }
2272
2273     @Override
2274     public int parse(DateTimeParseContext context, CharSequence text, int position) {
2275         // cache context before changed by decorated parser
2276         final boolean strict = context.isStrict();
2277         // parse

```

```

2276         if (position > text.length()) {
2277             throw new IndexOutOfBoundsException();
2278         }
2279         if (position == text.length()) {
2280             return ~position; // no more characters in the string
2281         }
2282         int endPos = position + padWidth;
2283         if (endPos > text.length()) {
2284             if (strict) {
2285                 return ~position; // not enough characters in the string to meet the parse
                                     width
2286             }
2287             endPos = text.length();
2288         }
2289         int pos = position;
2290         while (pos < endPos && context.charEquals(text.charAt(pos), padChar)) {
2291             pos++;
2292         }
2293         text = text.subSequence(0, endPos);
2294         int resultPos = printerParser.parse(context, text, pos);
2295         if (resultPos != endPos && strict) {
2296             return ~(position + pos); // parse of decorated field didn't parse to the end
2297         }
2298         return resultPos;
2299     }
2300
2301     @Override
2302     public String toString() {
2303         return "Pad(" + printerParser + "," + padWidth + (padChar == ' ' ? ")": ",'" + padChar
                + "')";
2304     }
2305 }
2306
2307 //-----
2308 /**
2309  * Enumeration to apply simple parse settings.
2310  */
2311 static enum SettingsParser implements DateTimePrinterParser {
2312     SENSITIVE,
2313     INSENSITIVE,
2314     STRICT,
2315     LENIENT;
2316
2317     @Override
2318     public boolean format(DateTimePrintContext context, StringBuilder buf) {
2319         return true; // nothing to do here
2320     }
2321
2322     @Override
2323     public int parse(DateTimeParseContext context, CharSequence text, int position) {
2324         // using ordinals to avoid javac synthetic inner class
2325         switch (ordinal()) {
2326             case 0: context.setCaseSensitive(true); break;

```

```

2327         case 1: context.setCaseSensitive(false); break;
2328         case 2: context.setStrict(true); break;
2329         case 3: context.setStrict(false); break;
2330     }
2331     return position;
2332 }
2333
2334 @Override
2335 public String toString() {
2336     // using ordinals to avoid javac synthetic inner class
2337     switch (ordinal()) {
2338         case 0: return "ParseCaseSensitive(true)";
2339         case 1: return "ParseCaseSensitive(false)";
2340         case 2: return "ParseStrict(true)";
2341         case 3: return "ParseStrict(false)";
2342     }
2343     throw new IllegalStateException("Unreachable");
2344 }
2345 }
2346
2347 //-----
2348 /**
2349  * Defaults a value into the parse if not currently present.
2350  */
2351 static class DefaultValueParser implements DateTimePrinterParser {
2352     private final TemporalField field;
2353     private final long value;
2354
2355     DefaultValueParser(TemporalField field, long value) {
2356         this.field = field;
2357         this.value = value;
2358     }
2359
2360     public boolean format(DateTimePrintContext context, StringBuilder buf) {
2361         return true;
2362     }
2363
2364     public int parse(DateTimeParseContext context, CharSequence text, int position) {
2365         if (context.getParsed(field) == null) {
2366             context.setParsedField(field, value, position, position);
2367         }
2368         return position;
2369     }
2370 }
2371
2372 //-----
2373 /**
2374  * Prints or parses a character literal.
2375  */
2376 static final class CharLiteralPrinterParser implements DateTimePrinterParser {
2377     private final char literal;
2378
2379     CharLiteralPrinterParser(char literal) {

```

```

2380         this.literal = literal;
2381     }
2382
2383     @Override
2384     public boolean format(DateTimePrintContext context, StringBuilder buf) {
2385         buf.append(literal);
2386         return true;
2387     }
2388
2389     @Override
2390     public int parse(DateTimeParseContext context, CharSequence text, int position) {
2391         int length = text.length();
2392         if (position == length) {
2393             return ~position;
2394         }
2395         char ch = text.charAt(position);
2396         if (ch != literal) {
2397             if (context.isCaseSensitive() ||
2398                 (Character.toUpperCase(ch) != Character.toUpperCase(literal) &&
2399                  Character.toLowerCase(ch) != Character.toLowerCase(literal))) {
2400                 return ~position;
2401             }
2402         }
2403         return position + 1;
2404     }
2405
2406     @Override
2407     public String toString() {
2408         if (literal == '\\') {
2409             return "'";
2410         }
2411         return "'" + literal + "'";
2412     }
2413 }
2414
2415 //-----
2416 /**
2417  * Prints or parses a string literal.
2418  */
2419 static final class StringLiteralPrinterParser implements DateTimePrinterParser {
2420     private final String literal;
2421
2422     StringLiteralPrinterParser(String literal) {
2423         this.literal = literal; // validated by caller
2424     }
2425
2426     @Override
2427     public boolean format(DateTimePrintContext context, StringBuilder buf) {
2428         buf.append(literal);
2429         return true;
2430     }
2431
2432     @Override

```

```

2433     public int parse(DateTimeParseContext context, CharSequence text, int position) {
2434         int length = text.length();
2435         if (position > length || position < 0) {
2436             throw new IndexOutOfBoundsException();
2437         }
2438         if (context.subSequenceEquals(text, position, literal, 0, literal.length()) == false) {
2439             return ~position;
2440         }
2441         return position + literal.length();
2442     }
2443
2444     @Override
2445     public String toString() {
2446         String converted = literal.replace("'", "");
2447         return "'" + converted + "'";
2448     }
2449 }
2450
2451 //-----
2452 /**
2453  * Prints and parses a numeric date-time field with optional padding.
2454  */
2455 static class NumberPrinterParser implements DateTimePrinterParser {
2456
2457     /**
2458      * Array of 10 to the power of n.
2459      */
2460     static final long[] EXCEED_POINTS = new long[] {
2461         0L,
2462         10L,
2463         100L,
2464         1000L,
2465         10000L,
2466         100000L,
2467         1000000L,
2468         10000000L,
2469         100000000L,
2470         1000000000L,
2471         10000000000L,
2472     };
2473
2474     final TemporalField field;
2475     final int minWidth;
2476     final int maxWidth;
2477     private final SignStyle signStyle;
2478     final int subsequentWidth;
2479
2480     /**
2481      * Constructor.
2482      *
2483      * @param field the field to format, not null
2484      * @param minWidth the minimum field width, from 1 to 19
2485      * @param maxWidth the maximum field width, from minWidth to 19

```



```

2486     * @param signStyle the positive/negative sign style, not null
2487     */
2488     NumberPrinterParser(TemporalField field, int minWidth, int maxWidth, SignStyle signStyle) {
2489         // validated by caller
2490         this.field = field;
2491         this.minWidth = minWidth;
2492         this.maxWidth = maxWidth;
2493         this.signStyle = signStyle;
2494         this.subsequentWidth = 0;
2495     }
2496
2497     /**
2498     * Constructor.
2499     *
2500     * @param field the field to format, not null
2501     * @param minWidth the minimum field width, from 1 to 19
2502     * @param maxWidth the maximum field width, from minWidth to 19
2503     * @param signStyle the positive/negative sign style, not null
2504     * @param subsequentWidth the width of subsequent non-negative numbers, 0 or greater,
2505     * -1 if fixed width due to active adjacent parsing
2506     */
2507     protected NumberPrinterParser(TemporalField field, int minWidth, int maxWidth, SignStyle
        signStyle, int subsequentWidth) {
2508         // validated by caller
2509         this.field = field;
2510         this.minWidth = minWidth;
2511         this.maxWidth = maxWidth;
2512         this.signStyle = signStyle;
2513         this.subsequentWidth = subsequentWidth;
2514     }
2515
2516     /**
2517     * Returns a new instance with fixed width flag set.
2518     *
2519     * @return a new updated printer-parser, not null
2520     */
2521     NumberPrinterParser withFixedWidth() {
2522         if (subsequentWidth == -1) {
2523             return this;
2524         }
2525         return new NumberPrinterParser(field, minWidth, maxWidth, signStyle, -1);
2526     }
2527
2528     /**
2529     * Returns a new instance with an updated subsequent width.
2530     *
2531     * @param subsequentWidth the width of subsequent non-negative numbers, 0 or greater
2532     * @return a new updated printer-parser, not null
2533     */
2534     NumberPrinterParser withSubsequentWidth(int subsequentWidth) {
2535         return new NumberPrinterParser(field, minWidth, maxWidth, signStyle, this.
            subsequentWidth + subsequentWidth);
2536     }

```

```

2537
2538     @Override
2539     public boolean format(DateTimePrintContext context, StringBuilder buf) {
2540         Long valueLong = context.getValue(field);
2541         if (valueLong == null) {
2542             return false;
2543         }
2544         long value = getValue(context, valueLong);
2545         DecimalStyle decimalStyle = context.getDecimalStyle();
2546         String str = (value == Long.MIN_VALUE ? "9223372036854775808" : Long.toString(Math.abs(
                value)));
2547         if (str.length() > maxWidth) {
2548             throw new DateTimeException("Field " + field +
2549                 " cannot be printed as the value " + value +
2550                 " exceeds the maximum print width of " + maxWidth);
2551         }
2552         str = decimalStyle.convertNumberToI18N(str);
2553
2554         if (value >= 0) {
2555             switch (signStyle) {
2556                 case EXCEEDS_PAD:
2557                     if (minWidth < 19 && value >= EXCEED_POINTS[minWidth]) {
2558                         buf.append(decimalStyle.getPositiveSign());
2559                     }
2560                     break;
2561                 case ALWAYS:
2562                     buf.append(decimalStyle.getPositiveSign());
2563                     break;
2564             }
2565         } else {
2566             switch (signStyle) {
2567                 case NORMAL:
2568                 case EXCEEDS_PAD:
2569                 case ALWAYS:
2570                     buf.append(decimalStyle.getNegativeSign());
2571                     break;
2572                 case NOT_NEGATIVE:
2573                     throw new DateTimeException("Field " + field +
2574                         " cannot be printed as the value " + value +
2575                         " cannot be negative according to the SignStyle");
2576             }
2577         }
2578         for (int i = 0; i < minWidth - str.length(); i++) {
2579             buf.append(decimalStyle.getZeroDigit());
2580         }
2581         buf.append(str);
2582         return true;
2583     }
2584
2585     /**
2586     * Gets the value to output.
2587     *
2588     * @param context the context

```

```

2589     * @param value the value of the field, not null
2590     * @return the value
2591     */
2592     long getValue(DateTimePrintContext context, long value) {
2593         return value;
2594     }
2595
2596     /**
2597     * For NumberPrinterParser, the width is fixed depending on the
2598     * minWidth, maxWidth, signStyle and whether subsequent fields are fixed.
2599     * @param context the context
2600     * @return true if the field is fixed width
2601     * @see DateTimeFormatterBuilder#appendValue(java.time.temporal.TemporalField, int)
2602     */
2603     boolean isFixedWidth(DateTimeParseContext context) {
2604         return subsequentWidth == -1 ||
2605             (subsequentWidth > 0 && minWidth == maxWidth && signStyle == SignStyle.NOT_NEGATIVE
2606             );
2607     }
2608
2609     @Override
2610     public int parse(DateTimeParseContext context, CharSequence text, int position) {
2611         int length = text.length();
2612         if (position == length) {
2613             return ~position;
2614         }
2615         char sign = text.charAt(position); // IOOBE if invalid position
2616         boolean negative = false;
2617         boolean positive = false;
2618         if (sign == context.getDecimalStyle().getPositiveSign()) {
2619             if (signStyle.parse(true, context.isStrict(), minWidth == maxWidth) == false) {
2620                 return ~position;
2621             }
2622             positive = true;
2623             position++;
2624         } else if (sign == context.getDecimalStyle().getNegativeSign()) {
2625             if (signStyle.parse(false, context.isStrict(), minWidth == maxWidth) == false) {
2626                 return ~position;
2627             }
2628             negative = true;
2629             position++;
2630         } else {
2631             if (signStyle == SignStyle.ALWAYS && context.isStrict()) {
2632                 return ~position;
2633             }
2634         }
2635         int effMinWidth = (context.isStrict() || isFixedWidth(context) ? minWidth : 1);
2636         int minEndPos = position + effMinWidth;
2637         if (minEndPos > length) {
2638             return ~position;
2639         }
2640         int effMaxWidth = (context.isStrict() || isFixedWidth(context) ? maxWidth : 9) + Math.
2641             max(subsequentWidth, 0);

```

```

2640     long total = 0;
2641     BigInteger totalBig = null;
2642     int pos = position;
2643     for (int pass = 0; pass < 2; pass++) {
2644         int maxEndPos = Math.min(pos + effMaxWidth, length);
2645         while (pos < maxEndPos) {
2646             char ch = text.charAt(pos++);
2647             int digit = context.getDecimalStyle().convertToDigit(ch);
2648             if (digit < 0) {
2649                 pos--;
2650                 if (pos < minEndPos) {
2651                     return ~(position); // need at least min width digits
2652                 }
2653                 break;
2654             }
2655             if ((pos - position) > 18) {
2656                 if (totalBig == null) {
2657                     totalBig = BigInteger.valueOf(total);
2658                 }
2659                 totalBig = totalBig.multiply(BigInteger.TEN).add(BigInteger.valueOf(digit))
                ;
2660             } else {
2661                 total = total * 10 + digit;
2662             }
2663         }
2664         if (subsequentWidth > 0 && pass == 0) {
2665             // re-parse now we know the correct width
2666             int parseLen = pos - position;
2667             effMaxWidth = Math.max(effMinWidth, parseLen - subsequentWidth);
2668             pos = position;
2669             total = 0;
2670             totalBig = null;
2671         } else {
2672             break;
2673         }
2674     }
2675     if (negative) {
2676         if (totalBig != null) {
2677             if (totalBig.equals(BigInteger.ZERO) && context.isStrict()) {
2678                 return ~(position - 1); // minus zero not allowed
2679             }
2680             totalBig = totalBig.negate();
2681         } else {
2682             if (total == 0 && context.isStrict()) {
2683                 return ~(position - 1); // minus zero not allowed
2684             }
2685             total = -total;
2686         }
2687     } else if (signStyle == SignStyle.EXCEEDS_PAD && context.isStrict()) {
2688         int parseLen = pos - position;
2689         if (positive) {
2690             if (parseLen <= minWidth) {
2691                 return ~(position - 1); // '+' only parsed if minWidth exceeded

```

```

2692     }
2693     } else {
2694         if (parseLen > minWidth) {
2695             return ~position; // '+' must be parsed if minWidth exceeded
2696         }
2697     }
2698 }
2699 if (totalBig != null) {
2700     if (totalBig.bitLength() > 63) {
2701         // overflow, parse 1 less digit
2702         totalBig = totalBig.divide(BigInteger.TEN);
2703         pos--;
2704     }
2705     return setValue(context, totalBig.longValue(), position, pos);
2706 }
2707 return setValue(context, total, position, pos);
2708 }
2709
2710 /**
2711  * Stores the value.
2712  *
2713  * @param context the context to store into, not null
2714  * @param value the value
2715  * @param errorPos the position of the field being parsed
2716  * @param successPos the position after the field being parsed
2717  * @return the new position
2718  */
2719 int setValue(DateTimeParseContext context, long value, int errorPos, int successPos) {
2720     return context.setParsedField(field, value, errorPos, successPos);
2721 }
2722
2723 @Override
2724 public String toString() {
2725     if (minWidth == 1 && maxWidth == 19 && signStyle == SignStyle.NORMAL) {
2726         return "Value(" + field + ")";
2727     }
2728     if (minWidth == maxWidth && signStyle == SignStyle.NOT_NEGATIVE) {
2729         return "Value(" + field + "," + minWidth + ")";
2730     }
2731     return "Value(" + field + "," + minWidth + "," + maxWidth + "," + signStyle + ")";
2732 }
2733 }
2734
2735 //-----
2736 /**
2737  * Prints and parses a reduced numeric date-time field.
2738  */
2739 static final class ReducedPrinterParser extends NumberPrinterParser {
2740     /**
2741      * The base date for reduced value parsing.
2742      */
2743     static final LocalDate BASE_DATE = LocalDate.of(2000, 1, 1);
2744 }

```

```

2745     private final int baseValue;
2746     private final ChronoLocalDate baseDate;
2747
2748     /**
2749      * Constructor.
2750      *
2751      * @param field the field to format, validated not null
2752      * @param minWidth the minimum field width, from 1 to 10
2753      * @param maxWidth the maximum field width, from 1 to 10
2754      * @param baseValue the base value
2755      * @param baseDate the base date
2756      */
2757     ReducedPrinterParser(TemporalField field, int minWidth, int maxWidth,
2758         int baseValue, ChronoLocalDate baseDate) {
2759         this(field, minWidth, maxWidth, baseValue, baseDate, 0);
2760         if (minWidth < 1 || minWidth > 10) {
2761             throw new IllegalArgumentException("The minWidth must be from 1 to 10 inclusive but
2762                 was " + minWidth);
2763         }
2764         if (maxWidth < 1 || maxWidth > 10) {
2765             throw new IllegalArgumentException("The maxWidth must be from 1 to 10 inclusive but
2766                 was " + minWidth);
2767         }
2768         if (maxWidth < minWidth) {
2769             throw new IllegalArgumentException("Maximum width must exceed or equal the minimum
2770                 width but " +
2771                 maxWidth + " < " + minWidth);
2772         }
2773         if (baseDate == null) {
2774             if (field.range().isValidValue(baseValue) == false) {
2775                 throw new IllegalArgumentException("The base value must be within the range of
2776                     the field");
2777             }
2778             if (((long) baseValue) + EXCEED_POINTS[maxWidth] > Integer.MAX_VALUE) {
2779                 throw new DateTimeException("Unable to add printer-parser as the range exceeds
2780                     the capacity of an int");
2781             }
2782         }
2783     }
2784
2785     /**
2786      * Constructor.
2787      * The arguments have already been checked.
2788      *
2789      * @param field the field to format, validated not null
2790      * @param minWidth the minimum field width, from 1 to 10
2791      * @param maxWidth the maximum field width, from 1 to 10
2792      * @param baseValue the base value
2793      * @param baseDate the base date
2794      * @param subsequentWidth the subsequentWidth for this instance
2795      */
2796     private ReducedPrinterParser(TemporalField field, int minWidth, int maxWidth,
2797         int baseValue, ChronoLocalDate baseDate, int subsequentWidth) {

```

```

2793     super(field, minWidth, maxWidth, SignStyle.NOT_NEGATIVE, subsequentWidth);
2794     this.baseValue = baseValue;
2795     this.baseDate = baseDate;
2796 }
2797
2798 @Override
2799 long getValue(DateTimePrintContext context, long value) {
2800     long absValue = Math.abs(value);
2801     int baseValue = this.baseValue;
2802     if (baseDate != null) {
2803         Chronology chrono = Chronology.from(context.getTemporal());
2804         baseValue = chrono.date(baseDate).get(field);
2805     }
2806     if (value >= baseValue && value < baseValue + EXCEED_POINTS[minWidth]) {
2807         // Use the reduced value if it fits in minWidth
2808         return absValue % EXCEED_POINTS[minWidth];
2809     }
2810     // Otherwise truncate to fit in maxWidth
2811     return absValue % EXCEED_POINTS[maxWidth];
2812 }
2813
2814 @Override
2815 int setValue(DateTimeParseContext context, long value, int errorPos, int successPos) {
2816     int baseValue = this.baseValue;
2817     if (baseDate != null) {
2818         Chronology chrono = context.getEffectiveChronology();
2819         baseValue = chrono.date(baseDate).get(field);
2820
2821         // In case the Chronology is changed later, add a callback when/if it changes
2822         final long initialValue = value;
2823         context.addChronoChangedListener(
2824             (_unused) -> {
2825                 /* Repeat the set of the field using the current Chronology
2826                  * The success/error position is ignored because the value is
2827                  * intentionally being overwritten.
2828                  */
2829                 setValue(context, initialValue, errorPos, successPos);
2830             });
2831     }
2832     int parseLen = successPos - errorPos;
2833     if (parseLen == minWidth && value >= 0) {
2834         long range = EXCEED_POINTS[minWidth];
2835         long lastPart = baseValue % range;
2836         long basePart = baseValue - lastPart;
2837         if (baseValue > 0) {
2838             value = basePart + value;
2839         } else {
2840             value = basePart - value;
2841         }
2842         if (value < baseValue) {
2843             value += range;
2844         }
2845     }

```

```

2846         return context.setParsedField(field, value, errorPos, successPos);
2847     }
2848
2849     /**
2850      * Returns a new instance with fixed width flag set.
2851      *
2852      * @return a new updated printer-parser, not null
2853      */
2854     @Override
2855     ReducedPrinterParser withFixedWidth() {
2856         if (subsequentWidth == -1) {
2857             return this;
2858         }
2859         return new ReducedPrinterParser(field, minWidth, maxWidth, baseValue, baseDate, -1);
2860     }
2861
2862     /**
2863      * Returns a new instance with an updated subsequent width.
2864      *
2865      * @param subsequentWidth the width of subsequent non-negative numbers, 0 or greater
2866      * @return a new updated printer-parser, not null
2867      */
2868     @Override
2869     ReducedPrinterParser withSubsequentWidth(int subsequentWidth) {
2870         return new ReducedPrinterParser(field, minWidth, maxWidth, baseValue, baseDate,
2871             this.subsequentWidth + subsequentWidth);
2872     }
2873
2874     /**
2875      * For a ReducedPrinterParser, fixed width is false if the mode is strict,
2876      * otherwise it is set as for NumberPrinterParser.
2877      * @param context the context
2878      * @return if the field is fixed width
2879      * @see DateTimeFormatterBuilder#appendValueReduced(java.time.temporal.TemporalField, int,
2880      *         int, int)
2881      */
2882     @Override
2883     boolean isFixedWidth(DateTimeParseContext context) {
2884         if (context.isStrict() == false) {
2885             return false;
2886         }
2887         return super.isFixedWidth(context);
2888     }
2889
2890     @Override
2891     public String toString() {
2892         return "ReducedValue(" + field + "," + minWidth + "," + maxWidth + "," + (baseDate !=
2893             null ? baseDate : baseValue) + ")";
2894     }
2895
2896     //-----
2897     /**

```



```

2897     * Prints and parses a numeric date-time field with optional padding.
2898     */
2899     static final class FractionPrinterParser implements DateTimePrinterParser {
2900         private final TemporalField field;
2901         private final int minWidth;
2902         private final int maxWidth;
2903         private final boolean decimalPoint;
2904
2905         /**
2906          * Constructor.
2907          *
2908          * @param field the field to output, not null
2909          * @param minWidth the minimum width to output, from 0 to 9
2910          * @param maxWidth the maximum width to output, from 0 to 9
2911          * @param decimalPoint whether to output the localized decimal point symbol
2912          */
2913         FractionPrinterParser(TemporalField field, int minWidth, int maxWidth, boolean decimalPoint
2914             ) {
2915             Objects.requireNonNull(field, "field");
2916             if (field.range().isFixed() == false) {
2917                 throw new IllegalArgumentException("Field must have a fixed set of values: " +
2918                     field);
2919             }
2920             if (minWidth < 0 || minWidth > 9) {
2921                 throw new IllegalArgumentException("Minimum width must be from 0 to 9 inclusive but
2922                     was " + minWidth);
2923             }
2924             if (maxWidth < 1 || maxWidth > 9) {
2925                 throw new IllegalArgumentException("Maximum width must be from 1 to 9 inclusive but
2926                     was " + maxWidth);
2927             }
2928             if (maxWidth < minWidth) {
2929                 throw new IllegalArgumentException("Maximum width must exceed or equal the minimum
2930                     width but " +
2931                     maxWidth + " < " + minWidth);
2932             }
2933             this.field = field;
2934             this.minWidth = minWidth;
2935             this.maxWidth = maxWidth;
2936             this.decimalPoint = decimalPoint;
2937         }
2938
2939         @Override
2940         public boolean format(DateTimePrintContext context, StringBuilder buf) {
2941             Long value = context.getValue(field);
2942             if (value == null) {
2943                 return false;
2944             }
2945             DecimalStyle decimalStyle = context.getDecimalStyle();
2946             BigDecimal fraction = convertToFraction(value);
2947             if (fraction.scale() == 0) { // scale is zero if value is zero
2948                 if (minWidth > 0) {
2949                     if (decimalPoint) {

```

```

2945         buf.append(decimalStyle.getDecimalSeparator());
2946     }
2947     for (int i = 0; i < minWidth; i++) {
2948         buf.append(decimalStyle.getZeroDigit());
2949     }
2950 }
2951 } else {
2952     int outputScale = Math.min(Math.max(fraction.scale(), minWidth), maxWidth);
2953     fraction = fraction.setScale(outputScale, RoundingMode.FLOOR);
2954     String str = fraction.toPlainString().substring(2);
2955     str = decimalStyle.convertNumberToI18N(str);
2956     if (decimalPoint) {
2957         buf.append(decimalStyle.getDecimalSeparator());
2958     }
2959     buf.append(str);
2960 }
2961 return true;
2962 }
2963
2964 @Override
2965 public int parse(DateTimeParseContext context, CharSequence text, int position) {
2966     int effectiveMin = (context.isStrict() ? minWidth : 0);
2967     int effectiveMax = (context.isStrict() ? maxWidth : 9);
2968     int length = text.length();
2969     if (position == length) {
2970         // valid if whole field is optional, invalid if minimum width
2971         return (effectiveMin > 0 ? ~position : position);
2972     }
2973     if (decimalPoint) {
2974         if (text.charAt(position) != context.getDecimalStyle().getDecimalSeparator()) {
2975             // valid if whole field is optional, invalid if minimum width
2976             return (effectiveMin > 0 ? ~position : position);
2977         }
2978         position++;
2979     }
2980     int minEndPos = position + effectiveMin;
2981     if (minEndPos > length) {
2982         return ~position; // need at least min width digits
2983     }
2984     int maxEndPos = Math.min(position + effectiveMax, length);
2985     int total = 0; // can use int because we are only parsing up to 9 digits
2986     int pos = position;
2987     while (pos < maxEndPos) {
2988         char ch = text.charAt(pos++);
2989         int digit = context.getDecimalStyle().convertToDigit(ch);
2990         if (digit < 0) {
2991             if (pos < minEndPos) {
2992                 return ~position; // need at least min width digits
2993             }
2994             pos--;
2995             break;
2996         }
2997         total = total * 10 + digit;

```

```

2998     }
2999     BigDecimal fraction = new BigDecimal(total).movePointLeft(pos - position);
3000     long value = convertFromFraction(fraction);
3001     return context.setParsedField(field, value, position, pos);
3002 }
3003
3004 /**
3005  * Converts a value for this field to a fraction between 0 and 1.
3006  * <p>
3007  * The fractional value is between 0 (inclusive) and 1 (exclusive).
3008  * It can only be returned if the {@link java.time.temporal.TemporalField#range() value
3009    range} is fixed.
3010  * The fraction is obtained by calculation from the field range using 9 decimal
3011  * places and a rounding mode of {@link RoundingMode#FLOOR FLOOR}.
3012  * The calculation is inaccurate if the values do not run continuously from smallest to
3013    largest.
3014  * <p>
3015  * For example, the second-of-minute value of 15 would be returned as 0.25,
3016  * assuming the standard definition of 60 seconds in a minute.
3017  *
3018  * @param value the value to convert, must be valid for this rule
3019  * @return the value as a fraction within the range, from 0 to 1, not null
3020  * @throws DateTimeException if the value cannot be converted to a fraction
3021  */
3022 private BigDecimal convertToFraction(long value) {
3023     ValueRange range = field.range();
3024     range.checkValidValue(value, field);
3025     BigDecimal minBD = BigDecimal.valueOf(range.getMinimum());
3026     BigDecimal rangeBD = BigDecimal.valueOf(range.getMaximum()).subtract(minBD).add(
3027         BigDecimal.ONE);
3028     BigDecimal valueBD = BigDecimal.valueOf(value).subtract(minBD);
3029     BigDecimal fraction = valueBD.divide(rangeBD, 9, RoundingMode.FLOOR);
3030     // stripTrailingZeros bug
3031     return fraction.compareTo(BigDecimal.ZERO) == 0 ? BigDecimal.ZERO : fraction.
3032         stripTrailingZeros();
3033 }
3034
3035 /**
3036  * Converts a fraction from 0 to 1 for this field to a value.
3037  * <p>
3038  * The fractional value must be between 0 (inclusive) and 1 (exclusive).
3039  * It can only be returned if the {@link java.time.temporal.TemporalField#range() value
3040    range} is fixed.
3041  * The value is obtained by calculation from the field range and a rounding
3042  * mode of {@link RoundingMode#FLOOR FLOOR}.
3043  * The calculation is inaccurate if the values do not run continuously from smallest to
3044    largest.
3045  * <p>
3046  * For example, the fractional second-of-minute of 0.25 would be converted to 15,
3047  * assuming the standard definition of 60 seconds in a minute.
3048  *
3049  * @param fraction the fraction to convert, not null
3050  * @return the value of the field, valid for this rule

```

```

3045     * @throws DateTimeException if the value cannot be converted
3046     */
3047     private long convertFromFraction(BigDecimal fraction) {
3048         ValueRange range = field.range();
3049         BigDecimal minBD = BigDecimal.valueOf(range.getMinimum());
3050         BigDecimal rangeBD = BigDecimal.valueOf(range.getMaximum()).subtract(minBD).add(
3051             BigDecimal.ONE);
3052         BigDecimal valueBD = fraction.multiply(rangeBD).setScale(0, RoundingMode.FLOOR).add(
3053             minBD);
3054         return valueBD.longValueExact();
3055     }
3056
3057     @Override
3058     public String toString() {
3059         String decimal = (decimalPoint ? ",DecimalPoint" : "");
3060         return "Fraction(" + field + "," + minWidth + "," + maxWidth + decimal + ")";
3061     }
3062 }
3063
3064 //-----
3065 /**
3066  * Prints or parses field text.
3067  */
3068 static final class TextPrinterParser implements DateTimePrinterParser {
3069     private final TemporalField field;
3070     private final TextStyle textStyle;
3071     private final DateTimeTextProvider provider;
3072
3073     /**
3074      * The cached number printer parser.
3075      * Immutable and volatile, so no synchronization needed.
3076      */
3077     private volatile NumberPrinterParser numberPrinterParser;
3078
3079     /**
3080      * Constructor.
3081      *
3082      * @param field the field to output, not null
3083      * @param textStyle the text style, not null
3084      * @param provider the text provider, not null
3085      */
3086     TextPrinterParser(TemporalField field, TextStyle textStyle, DateTimeTextProvider provider)
3087     {
3088         // validated by caller
3089         this.field = field;
3090         this.textStyle = textStyle;
3091         this.provider = provider;
3092     }
3093
3094     @Override
3095     public boolean format(DateTimePrintContext context, StringBuilder buf) {
3096         Long value = context.getValue(field);
3097         if (value == null) {
3098             return false;
3099         }

```

```

3095     }
3096     String text;
3097     Chronology chrono = context.getTemporal().query(TemporalQueries.chronology());
3098     if (chrono == null || chrono == IsoChronology.INSTANCE) {
3099         text = provider.getText(field, value, textStyle, context.getLocale());
3100     } else {
3101         text = provider.getText(chrono, field, value, textStyle, context.getLocale());
3102     }
3103     if (text == null) {
3104         return numberPrinterParser().format(context, buf);
3105     }
3106     buf.append(text);
3107     return true;
3108 }
3109
3110 @Override
3111 public int parse(DateTimeParseContext context, CharSequence parseText, int position) {
3112     int length = parseText.length();
3113     if (position < 0 || position > length) {
3114         throw new IndexOutOfBoundsException();
3115     }
3116     TextStyle style = (context.isStrict() ? textStyle : null);
3117     Chronology chrono = context.getEffectiveChronology();
3118     Iterator<Entry<String, Long>> it;
3119     if (chrono == null || chrono == IsoChronology.INSTANCE) {
3120         it = provider.getTextIterator(field, style, context.getLocale());
3121     } else {
3122         it = provider.getTextIterator(chrono, field, style, context.getLocale());
3123     }
3124     if (it != null) {
3125         while (it.hasNext()) {
3126             Entry<String, Long> entry = it.next();
3127             String itText = entry.getKey();
3128             if (context.subSequenceEquals(itText, 0, parseText, position, itText.length()))
3129                 return context.setParsedField(field, entry.getValue(), position, position +
3130                     itText.length());
3131         }
3132         if (context.isStrict()) {
3133             return ~position;
3134         }
3135     }
3136     return numberPrinterParser().parse(context, parseText, position);
3137 }
3138
3139 /**
3140  * Create and cache a number printer parser.
3141  * @return the number printer parser for this field, not null
3142  */
3143 private NumberPrinterParser numberPrinterParser() {
3144     if (numberPrinterParser == null) {
3145         numberPrinterParser = new NumberPrinterParser(field, 1, 19, SignStyle.NORMAL);

```

```

3146     }
3147     return numberPrinterParser;
3148 }
3149
3150 @Override
3151 public String toString() {
3152     if (textStyle == TextStyle.FULL) {
3153         return "Text(" + field + ")";
3154     }
3155     return "Text(" + field + "," + textStyle + ")";
3156 }
3157 }
3158
3159 //-----
3160 /**
3161  * Prints or parses an ISO-8601 instant.
3162  */
3163 static final class InstantPrinterParser implements DateTimePrinterParser {
3164     // days in a 400 year cycle = 146097
3165     // days in a 10,000 year cycle = 146097 * 25
3166     // seconds per day = 86400
3167     private static final long SECONDS_PER_10000_YEARS = 146097L * 25L * 86400L;
3168     private static final long SECONDS_0000_TO_1970 = ((146097L * 5L) - (30L * 365L + 7L)) *
3169         86400L;
3170     private final int fractionalDigits;
3171
3172     InstantPrinterParser(int fractionalDigits) {
3173         this.fractionalDigits = fractionalDigits;
3174     }
3175
3176 @Override
3177 public boolean format(DateTimePrintContext context, StringBuilder buf) {
3178     // use INSTANT_SECONDS, thus this code is not bound by Instant.MAX
3179     Long inSecs = context.getValue(INSTANT_SECONDS);
3180     Long inNanos = null;
3181     if (context.getTemporal().isSupported(NANO_OF_SECOND)) {
3182         inNanos = context.getTemporal().getLong(NANO_OF_SECOND);
3183     }
3184     if (inSecs == null) {
3185         return false;
3186     }
3187     long inSec = inSecs;
3188     int inNano = NANO_OF_SECOND.checkValidIntValue(inNanos != null ? inNanos : 0);
3189     // format mostly using LocalDateTime.toString
3190     if (inSec >= -SECONDS_0000_TO_1970) {
3191         // current era
3192         long zeroSecs = inSec - SECONDS_PER_10000_YEARS + SECONDS_0000_TO_1970;
3193         long hi = Math.floorDiv(zeroSecs, SECONDS_PER_10000_YEARS) + 1;
3194         long lo = Math.floorMod(zeroSecs, SECONDS_PER_10000_YEARS);
3195         LocalDateTime ldt = LocalDateTime.ofEpochSecond(lo - SECONDS_0000_TO_1970, 0,
3196             ZoneOffset.UTC);
3197         if (hi > 0) {
3198             buf.append('+').append(hi);

```

```

3197     }
3198     buf.append(ldt);
3199     if (ldt.getSecond() == 0) {
3200         buf.append(":00");
3201     }
3202     } else {
3203         // before current era
3204         long zeroSecs = inSec + SECONDS_0000_TO_1970;
3205         long hi = zeroSecs / SECONDS_PER_10000_YEARS;
3206         long lo = zeroSecs % SECONDS_PER_10000_YEARS;
3207         LocalDateTime ldt = LocalDateTime.ofEpochSecond(lo - SECONDS_0000_TO_1970, 0,
3208             ZoneOffset.UTC);
3209         int pos = buf.length();
3210         buf.append(ldt);
3211         if (ldt.getSecond() == 0) {
3212             buf.append(":00");
3213         }
3214         if (hi < 0) {
3215             if (ldt.getYear() == -10_000) {
3216                 buf.replace(pos, pos + 2, Long.toString(hi - 1));
3217             } else if (lo == 0) {
3218                 buf.insert(pos, hi);
3219             } else {
3220                 buf.insert(pos + 1, Math.abs(hi));
3221             }
3222         }
3223     }
3224     // add fraction
3225     if ((fractionalDigits < 0 && inNano > 0) || fractionalDigits > 0) {
3226         buf.append('.');
3227         int div = 100_000_000;
3228         for (int i = 0; ((fractionalDigits == -1 && inNano > 0) ||
3229             (fractionalDigits == -2 && (inNano > 0 || (i % 3) != 0)) ||
3230             i < fractionalDigits); i++) {
3231             int digit = inNano / div;
3232             buf.append((char) (digit + '0'));
3233             inNano = inNano - (digit * div);
3234             div = div / 10;
3235         }
3236     }
3237     buf.append('Z');
3238     return true;
3239 }
3240
3241 @Override
3242 public int parse(DateTimeParseContext context, CharSequence text, int position) {
3243     // new context to avoid overwriting fields like year/month/day
3244     int minDigits = (fractionalDigits < 0 ? 0 : fractionalDigits);
3245     int maxDigits = (fractionalDigits < 0 ? 9 : fractionalDigits);
3246     CompositePrinterParser parser = new DateTimeFormatterBuilder()
3247         .append(DateTimeFormatter.ISO_LOCAL_DATE).appendLiteral('T')
3248         .appendValue(HOUR_OF_DAY, 2).appendLiteral(':')
3249         .appendValue(MINUTE_OF_HOUR, 2).appendLiteral(':')

```

```

3249         .appendValue(SECOND_OF_MINUTE, 2)
3250         .appendFraction(NANO_OF_SECOND, minDigits, maxDigits, true)
3251         .appendLiteral('Z')
3252         .toFormatter().toPrinterParser(false);
3253     DateTimeParseContext newContext = context.copy();
3254     int pos = parser.parse(newContext, text, position);
3255     if (pos < 0) {
3256         return pos;
3257     }
3258     // parser restricts most fields to 2 digits, so definitely int
3259     // correctly parsed nano is also guaranteed to be valid
3260     long yearParsed = newContext.getParsed(YEAR);
3261     int month = newContext.getParsed(MONTH_OF_YEAR).intValue();
3262     int day = newContext.getParsed(DAY_OF_MONTH).intValue();
3263     int hour = newContext.getParsed(HOUR_OF_DAY).intValue();
3264     int min = newContext.getParsed(MINUTE_OF_HOUR).intValue();
3265     Long secVal = newContext.getParsed(SECOND_OF_MINUTE);
3266     Long nanoVal = newContext.getParsed(NANO_OF_SECOND);
3267     int sec = (secVal != null ? secVal.intValue() : 0);
3268     int nano = (nanoVal != null ? nanoVal.intValue() : 0);
3269     int days = 0;
3270     if (hour == 24 && min == 0 && sec == 0 && nano == 0) {
3271         hour = 0;
3272         days = 1;
3273     } else if (hour == 23 && min == 59 && sec == 60) {
3274         context.setParsedLeapSecond();
3275         sec = 59;
3276     }
3277     int year = (int) yearParsed % 10_000;
3278     long instantSecs;
3279     try {
3280         LocalDateTime ldt = LocalDateTime.of(year, month, day, hour, min, sec, 0).plusDays(
3281             days);
3282         instantSecs = ldt.toEpochSecond(ZoneOffset.UTC);
3283         instantSecs += Math.multiplyExact(yearParsed / 10_000L, SECONDS_PER_10000_YEARS);
3284     } catch (RuntimeException ex) {
3285         return ~position;
3286     }
3287     int successPos = pos;
3288     successPos = context.setParsedField(INSTANT_SECONDS, instantSecs, position, successPos)
3289         ;
3290     return context.setParsedField(NANO_OF_SECOND, nano, position, successPos);
3291 }
3292
3293 @Override
3294 public String toString() {
3295     return "Instant()";
3296 }
3297 }
3298
3299 //-----
3300 /**
3301  * Prints or parses an offset ID.

```



```

3300     */
3301     static final class OffsetIdPrinterParser implements DateTimePrinterParser {
3302         static final String[] PATTERNS = new String[] {
3303             "+HH", "+HHmm", "+HH:mm", "+HHMM", "+HH:MM", "+HHMMss", "+HH:MM:ss", "+HHMMSS", "+HH:MM
3304             :SS",
3305         }; // order used in pattern builder
3306         static final OffsetIdPrinterParser INSTANCE_ID_Z = new OffsetIdPrinterParser("+HH:MM:ss", "
3307             Z");
3308         static final OffsetIdPrinterParser INSTANCE_ID_ZERO = new OffsetIdPrinterParser("+HH:MM:ss"
3309             , "0");
3310
3311         private final String noOffsetText;
3312         private final int type;
3313
3314         /**
3315          * Constructor.
3316          *
3317          * @param pattern the pattern
3318          * @param noOffsetText the text to use for UTC, not null
3319          */
3320         OffsetIdPrinterParser(String pattern, String noOffsetText) {
3321             Objects.requireNonNull(pattern, "pattern");
3322             Objects.requireNonNull(noOffsetText, "noOffsetText");
3323             this.type = checkPattern(pattern);
3324             this.noOffsetText = noOffsetText;
3325         }
3326
3327         private int checkPattern(String pattern) {
3328             for (int i = 0; i < PATTERNS.length; i++) {
3329                 if (PATTERNS[i].equals(pattern)) {
3330                     return i;
3331                 }
3332             }
3333             throw new IllegalArgumentException("Invalid zone offset pattern: " + pattern);
3334         }
3335
3336         @Override
3337         public boolean format(DateTimePrintContext context, StringBuilder buf) {
3338             Long offsetSecs = context.getValue(OFFSET_SECONDS);
3339             if (offsetSecs == null) {
3340                 return false;
3341             }
3342             int totalSecs = Math.toIntExact(offsetSecs);
3343             if (totalSecs == 0) {
3344                 buf.append(noOffsetText);
3345             } else {
3346                 int absHours = Math.abs((totalSecs / 3600) % 100); // anything larger than 99
3347                     silently dropped
3348                 int absMinutes = Math.abs((totalSecs / 60) % 60);
3349                 int absSeconds = Math.abs(totalSecs % 60);
3350                 int bufPos = buf.length();
3351                 int output = absHours;
3352                 buf.append(totalSecs < 0 ? "-" : "+")

```

```

3349         .append((char) (absHours / 10 + '0')).append((char) (absHours % 10 + '0'));
3350     if (type >= 3 || (type >= 1 && absMinutes > 0)) {
3351         buf.append((type % 2) == 0 ? ":" : "");
3352         .append((char) (absMinutes / 10 + '0')).append((char) (absMinutes % 10 + '0'
3353             ));
3354         output += absMinutes;
3355         if (type >= 7 || (type >= 5 && absSeconds > 0)) {
3356             buf.append((type % 2) == 0 ? ":" : "");
3357             .append((char) (absSeconds / 10 + '0')).append((char) (absSeconds % 10
3358                 + '0'));
3359             output += absSeconds;
3360         }
3361     }
3362     if (output == 0) {
3363         buf.setLength(bufPos);
3364         buf.append(noOffsetText);
3365     }
3366     return true;
3367 }
3368
3369 @Override
3370 public int parse(DateTimeParseContext context, CharSequence text, int position) {
3371     int length = text.length();
3372     int noOffsetLen = noOffsetText.length();
3373     if (noOffsetLen == 0) {
3374         if (position == length) {
3375             return context.setParsedField(OFFSET_SECONDS, 0, position, position);
3376         }
3377     } else {
3378         if (position == length) {
3379             return ~position;
3380         }
3381         if (context.subSequenceEquals(text, position, noOffsetText, 0, noOffsetLen)) {
3382             return context.setParsedField(OFFSET_SECONDS, 0, position, position +
3383                 noOffsetLen);
3384         }
3385     }
3386
3387     // parse normal plus/minus offset
3388     char sign = text.charAt(position); // IOOBE if invalid position
3389     if (sign == '+' || sign == '-') {
3390         // starts
3391         int negative = (sign == '-' ? -1 : 1);
3392         int[] array = new int[4];
3393         array[0] = position + 1;
3394         if ((parseNumber(array, 1, text, true) ||
3395             parseNumber(array, 2, text, type >= 3) ||
3396             parseNumber(array, 3, text, false)) == false) {
3397             // success
3398             long offsetSecs = negative * (array[1] * 3600L + array[2] * 60L + array[3]);
3399             return context.setParsedField(OFFSET_SECONDS, offsetSecs, position, array[0]);
3400         }
3401     }

```

```

3399     }
3400     // handle special case of empty no offset text
3401     if (noOffsetLen == 0) {
3402         return context.setParsedField(OFFSET_SECONDS, 0, position, position + noOffsetLen);
3403     }
3404     return ~position;
3405 }
3406
3407 /**
3408  * Parse a two digit zero-prefixed number.
3409  *
3410  * @param array the array of parsed data, 0=pos,1=hours,2=mins,3=secs, not null
3411  * @param arrayIndex the index to parse the value into
3412  * @param parseText the offset ID, not null
3413  * @param required whether this number is required
3414  * @return true if an error occurred
3415  */
3416 private boolean parseNumber(int[] array, int arrayIndex, CharSequence parseText, boolean
    required) {
3417     if ((type + 3) / 2 < arrayIndex) {
3418         return false; // ignore seconds/minutes
3419     }
3420     int pos = array[0];
3421     if ((type % 2) == 0 && arrayIndex > 1) {
3422         if (pos + 1 > parseText.length() || parseText.charAt(pos) != ':') {
3423             return required;
3424         }
3425         pos++;
3426     }
3427     if (pos + 2 > parseText.length()) {
3428         return required;
3429     }
3430     char ch1 = parseText.charAt(pos++);
3431     char ch2 = parseText.charAt(pos++);
3432     if (ch1 < '0' || ch1 > '9' || ch2 < '0' || ch2 > '9') {
3433         return required;
3434     }
3435     int value = (ch1 - 48) * 10 + (ch2 - 48);
3436     if (value < 0 || value > 59) {
3437         return required;
3438     }
3439     array[arrayIndex] = value;
3440     array[0] = pos;
3441     return false;
3442 }
3443
3444 @Override
3445 public String toString() {
3446     String converted = noOffsetText.replace("'", "");
3447     return "Offset(" + PATTERNS[type] + ",'" + converted + "'";
3448 }
3449 }
3450

```

```

3451 //-----
3452 /**
3453  * Prints or parses an offset ID.
3454  */
3455 static final class LocalizedOffsetIdPrinterParser implements DateTimePrinterParser {
3456     private final TextStyle style;
3457
3458     /**
3459      * Constructor.
3460      *
3461      * @param style the style, not null
3462      */
3463     LocalizedOffsetIdPrinterParser(TextStyle style) {
3464         this.style = style;
3465     }
3466
3467     private static StringBuilder appendHMS(StringBuilder buf, int t) {
3468         return buf.append((char)(t / 10 + '0'))
3469             .append((char)(t % 10 + '0'));
3470     }
3471
3472     @Override
3473     public boolean format(DateTimePrintContext context, StringBuilder buf) {
3474         Long offsetSecs = context.getValue(OFFSET_SECONDS);
3475         if (offsetSecs == null) {
3476             return false;
3477         }
3478         String gmtText = "GMT"; // TODO: get localized version of 'GMT'
3479         if (gmtText != null) {
3480             buf.append(gmtText);
3481         }
3482         int totalSecs = Math.toIntExact(offsetSecs);
3483         if (totalSecs != 0) {
3484             int absHours = Math.abs((totalSecs / 3600) % 100); // anything larger than 99
3485                 silently dropped
3486             int absMinutes = Math.abs((totalSecs / 60) % 60);
3487             int absSeconds = Math.abs(totalSecs % 60);
3488             buf.append(totalSecs < 0 ? "-" : "+");
3489             if (style == TextStyle.FULL) {
3490                 appendHMS(buf, absHours);
3491                 buf.append(':');
3492                 appendHMS(buf, absMinutes);
3493                 if (absSeconds != 0) {
3494                     buf.append(':');
3495                     appendHMS(buf, absSeconds);
3496                 }
3497             } else {
3498                 if (absHours >= 10) {
3499                     buf.append((char)(absHours / 10 + '0'));
3500                 }
3501                 buf.append((char)(absHours % 10 + '0'));
3502                 if (absMinutes != 0 || absSeconds != 0) {
3503                     buf.append(':');

```

```

3503         appendHMS(buf, absMinutes);
3504         if (absSeconds != 0) {
3505             buf.append(':');
3506             appendHMS(buf, absSeconds);
3507         }
3508     }
3509 }
3510 }
3511 return true;
3512 }
3513
3514 int getDigit(CharSequence text, int position) {
3515     char c = text.charAt(position);
3516     if (c < '0' || c > '9') {
3517         return -1;
3518     }
3519     return c - '0';
3520 }
3521
3522 @Override
3523 public int parse(DateTimeParseContext context, CharSequence text, int position) {
3524     int pos = position;
3525     int end = pos + text.length();
3526     String gmtText = "GMT"; // TODO: get localized version of 'GMT'
3527     if (gmtText != null) {
3528         if (!context.subSequenceEquals(text, pos, gmtText, 0, gmtText.length())) {
3529             return ~position;
3530         }
3531         pos += gmtText.length();
3532     }
3533     // parse normal plus/minus offset
3534     int negative = 0;
3535     if (pos == end) {
3536         return context.setParsedField(OFFSET_SECONDS, 0, position, pos);
3537     }
3538     char sign = text.charAt(pos); // IOOBE if invalid position
3539     if (sign == '+') {
3540         negative = 1;
3541     } else if (sign == '-') {
3542         negative = -1;
3543     } else {
3544         return context.setParsedField(OFFSET_SECONDS, 0, position, pos);
3545     }
3546     pos++;
3547     int h = 0;
3548     int m = 0;
3549     int s = 0;
3550     if (style == TextStyle.FULL) {
3551         int h1 = getDigit(text, pos++);
3552         int h2 = getDigit(text, pos++);
3553         if (h1 < 0 || h2 < 0 || text.charAt(pos++) != ':') {
3554             return ~position;
3555         }

```

```

3556         h = h1 * 10 + h2;
3557         int m1 = getDigit(text, pos++);
3558         int m2 = getDigit(text, pos++);
3559         if (m1 < 0 || m2 < 0) {
3560             return ~position;
3561         }
3562         m = m1 * 10 + m2;
3563         if (pos + 2 < end && text.charAt(pos) == ':') {
3564             int s1 = getDigit(text, pos + 1);
3565             int s2 = getDigit(text, pos + 2);
3566             if (s1 >= 0 && s2 >= 0) {
3567                 s = s1 * 10 + s2;
3568                 pos += 3;
3569             }
3570         }
3571     } else {
3572         h = getDigit(text, pos++);
3573         if (h < 0) {
3574             return ~position;
3575         }
3576         if (pos < end) {
3577             int h2 = getDigit(text, pos);
3578             if (h2 >= 0) {
3579                 h = h * 10 + h2;
3580                 pos++;
3581             }
3582             if (pos + 2 < end && text.charAt(pos) == ':') {
3583                 if (pos + 2 < end && text.charAt(pos) == ':') {
3584                     int m1 = getDigit(text, pos + 1);
3585                     int m2 = getDigit(text, pos + 2);
3586                     if (m1 >= 0 && m2 >= 0) {
3587                         m = m1 * 10 + m2;
3588                         pos += 3;
3589                         if (pos + 2 < end && text.charAt(pos) == ':') {
3590                             int s1 = getDigit(text, pos + 1);
3591                             int s2 = getDigit(text, pos + 2);
3592                             if (s1 >= 0 && s2 >= 0) {
3593                                 s = s1 * 10 + s2;
3594                                 pos += 3;
3595                             }
3596                         }
3597                     }
3598                 }
3599             }
3600         }
3601     }
3602     long offsetSecs = negative * (h * 3600L + m * 60L + s);
3603     return context.setParsedField(OFFSET_SECONDS, offsetSecs, position, pos);
3604 }
3605
3606 @Override
3607 public String toString() {
3608     return "LocalizedOffset(" + style + ")";

```

```

3609     }
3610 }
3611
3612 //-----
3613 /**
3614  * Prints or parses a zone ID.
3615  */
3616 static final class ZoneTextPrinterParser extends ZoneIdPrinterParser {
3617
3618     /** The text style to output. */
3619     private final TextStyle textStyle;
3620
3621     /** The preferred zoneid map */
3622     private Set<String> preferredZones;
3623
3624     ZoneTextPrinterParser(TextStyle textStyle, Set<ZoneId> preferredZones) {
3625         super(TemporalQueries.zone(), "ZoneText(" + textStyle + ")");
3626         this.textStyle = Objects.requireNonNull(textStyle, "textStyle");
3627         if (preferredZones != null && preferredZones.size() != 0) {
3628             this.preferredZones = new HashSet<>();
3629             for (ZoneId id : preferredZones) {
3630                 this.preferredZones.add(id.getId());
3631             }
3632         }
3633     }
3634
3635     private static final int STD = 0;
3636     private static final int DST = 1;
3637     private static final int GENERIC = 2;
3638     private static final Map<String, SoftReference<Map<Locale, String[]>>> cache =
3639         new ConcurrentHashMap<>();
3640
3641     private String getDisplayName(String id, int type, Locale locale) {
3642         if (textStyle == TextStyle.NARROW) {
3643             return null;
3644         }
3645         String[] names;
3646         SoftReference<Map<Locale, String[]>> ref = cache.get(id);
3647         Map<Locale, String[]> perLocale = null;
3648         if (ref == null || (perLocale = ref.get()) == null ||
3649             (names = perLocale.get(locale)) == null) {
3650             names = TimeZoneNameUtility.retrieveDisplayNames(id, locale);
3651             if (names == null) {
3652                 return null;
3653             }
3654             names = Arrays.copyOfRange(names, 0, 7);
3655             names[5] =
3656                 TimeZoneNameUtility.retrieveGenericDisplayName(id, TimeZone.LONG, locale);
3657             if (names[5] == null) {
3658                 names[5] = names[0]; // use the id
3659             }
3660             names[6] =
3661                 TimeZoneNameUtility.retrieveGenericDisplayName(id, TimeZone.SHORT, locale);

```

```

3662         if (names[6] == null) {
3663             names[6] = names[0];
3664         }
3665         if (perLocale == null) {
3666             perLocale = new ConcurrentHashMap<>();
3667         }
3668         perLocale.put(locale, names);
3669         cache.put(id, new SoftReference<>(perLocale));
3670     }
3671     switch (type) {
3672     case STD:
3673         return names[textStyle.zoneNameStyleIndex() + 1];
3674     case DST:
3675         return names[textStyle.zoneNameStyleIndex() + 3];
3676     }
3677     return names[textStyle.zoneNameStyleIndex() + 5];
3678 }
3679
3680 @Override
3681 public boolean format(DateTimePrintContext context, StringBuilder buf) {
3682     ZoneId zone = context.getValue(TemporalQueries.zoneId());
3683     if (zone == null) {
3684         return false;
3685     }
3686     String zname = zone.getId();
3687     if (!(zone instanceof ZoneOffset)) {
3688         TemporalAccessor dt = context.getTemporal();
3689         String name = getDisplayName(zname,
3690                                     dt.isSupported(ChronoField.INSTANT_SECONDS)
3691                                     ? (zone.getRules().isDaylightSavings(Instant.from(dt))
3692                                        ? DST : STD)
3693                                     : GENERIC,
3694                                     context.getLocale());
3695         if (name != null) {
3696             zname = name;
3697         }
3698     }
3699     buf.append(zname);
3700     return true;
3701 }
3702
3703 // cache per instance for now
3704 private final Map<Locale, Entry<Integer, SoftReference<PrefixTree>>>
3705     cachedTree = new HashMap<>();
3706 private final Map<Locale, Entry<Integer, SoftReference<PrefixTree>>>
3707     cachedTreeCI = new HashMap<>();
3708
3709 @Override
3710 protected PrefixTree getTree(DateTimeParseContext context) {
3711     if (textStyle == TextStyle.NARROW) {
3712         return super.getTree(context);
3713     }
3714     Locale locale = context.getLocale();

```



```

3714         boolean isCaseSensitive = context.isCaseSensitive();
3715         Set<String> regionIds = ZoneRulesProvider.getAvailableZoneIds();
3716         int regionIdsSize = regionIds.size();
3717
3718         Map<Locale, Entry<Integer, SoftReference<PrefixTree>>> cached =
3719             isCaseSensitive ? cachedTree : cachedTreeCI;
3720
3721         Entry<Integer, SoftReference<PrefixTree>> entry = null;
3722         PrefixTree tree = null;
3723         String[] [] zoneStrings = null;
3724         if ((entry = cached.get(locale)) == null ||
3725             (entry.getKey() != regionIdsSize ||
3726              (tree = entry.getValue().get()) == null)) {
3727             tree = PrefixTree.newTree(context);
3728             zoneStrings = TimeZoneNameUtility.getZoneStrings(locale);
3729             for (String[] names : zoneStrings) {
3730                 String zid = names[0];
3731                 if (!regionIds.contains(zid)) {
3732                     continue;
3733                 }
3734                 tree.add(zid, zid); // don't convert zid -> metazone
3735                 zid = ZoneName.toZid(zid, locale);
3736                 int i = textStyle == TextStyle.FULL ? 1 : 2;
3737                 for (; i < names.length; i += 2) {
3738                     tree.add(names[i], zid);
3739                 }
3740             }
3741             // if we have a set of preferred zones, need a copy and
3742             // add the preferred zones again to overwrite
3743             if (preferredZones != null) {
3744                 for (String[] names : zoneStrings) {
3745                     String zid = names[0];
3746                     if (!preferredZones.contains(zid) || !regionIds.contains(zid)) {
3747                         continue;
3748                     }
3749                     int i = textStyle == TextStyle.FULL ? 1 : 2;
3750                     for (; i < names.length; i += 2) {
3751                         tree.add(names[i], zid);
3752                     }
3753                 }
3754             }
3755             cached.put(locale, new SimpleImmutableEntry<>(regionIdsSize, new SoftReference<>(
3756                 tree)));
3757         }
3758         return tree;
3759     }
3760
3761     //-----
3762     /**
3763      * Prints or parses a zone ID.
3764      */
3765     static class ZoneIdPrinterParser implements DateTimePrinterParser {

```

```

3766     private final TemporalQuery<ZoneId> query;
3767     private final String description;
3768
3769     ZoneIdPrinterParser(TemporalQuery<ZoneId> query, String description) {
3770         this.query = query;
3771         this.description = description;
3772     }
3773
3774     @Override
3775     public boolean format(DateTimePrintContext context, StringBuilder buf) {
3776         ZoneId zone = context.getValue(query);
3777         if (zone == null) {
3778             return false;
3779         }
3780         buf.append(zone.getId());
3781         return true;
3782     }
3783
3784     /**
3785      * The cached tree to speed up parsing.
3786      */
3787     private static volatile Entry<Integer, PrefixTree> cachedPrefixTree;
3788     private static volatile Entry<Integer, PrefixTree> cachedPrefixTreeCI;
3789
3790     protected PrefixTree getTree(DateTimeParseContext context) {
3791         // prepare parse tree
3792         Set<String> regionIds = ZoneRulesProvider.getAvailableZoneIds();
3793         final int regionIdsSize = regionIds.size();
3794         Entry<Integer, PrefixTree> cached = context.isCaseSensitive()
3795             ? cachedPrefixTree : cachedPrefixTreeCI;
3796         if (cached == null || cached.getKey() != regionIdsSize) {
3797             synchronized (this) {
3798                 cached = context.isCaseSensitive() ? cachedPrefixTree : cachedPrefixTreeCI;
3799                 if (cached == null || cached.getKey() != regionIdsSize) {
3800                     cached = new SimpleImmutableEntry<>(regionIdsSize, PrefixTree.newTree(
3801                         regionIds, context));
3802                     if (context.isCaseSensitive()) {
3803                         cachedPrefixTree = cached;
3804                     } else {
3805                         cachedPrefixTreeCI = cached;
3806                     }
3807                 }
3808             }
3809             return cached.getValue();
3810         }
3811
3812     /**
3813      * This implementation looks for the longest matching string.
3814      * For example, parsing Etc/GMT-2 will return Etc/GMC-2 rather than just
3815      * Etc/GMC although both are valid.
3816      */
3817     @Override

```

```

3818     public int parse(DateTimeParseContext context, CharSequence text, int position) {
3819         int length = text.length();
3820         if (position > length) {
3821             throw new IndexOutOfBoundsException();
3822         }
3823         if (position == length) {
3824             return ~position;
3825         }
3826
3827         // handle fixed time-zone IDs
3828         char nextChar = text.charAt(position);
3829         if (nextChar == '+' || nextChar == '-') {
3830             return parseOffsetBased(context, text, position, position, OffsetIdPrinterParser.
3831                                     INSTANCE_ID_Z);
3832         } else if (length >= position + 2) {
3833             char nextNextChar = text.charAt(position + 1);
3834             if (context.charEquals(nextChar, 'U') && context.charEquals(nextNextChar, 'T')) {
3835                 if (length >= position + 3 && context.charEquals(text.charAt(position + 2), 'C'
3836                     )) {
3837                     return parseOffsetBased(context, text, position, position + 3,
3838                                             OffsetIdPrinterParser.INSTANCE_ID_ZERO);
3839                 }
3840                 return parseOffsetBased(context, text, position, position + 2,
3841                                         OffsetIdPrinterParser.INSTANCE_ID_ZERO);
3842             } else if (context.charEquals(nextChar, 'G') && length >= position + 3 &&
3843                 context.charEquals(nextNextChar, 'M') && context.charEquals(text.charAt(
3844                     position + 2), 'T')) {
3845                 return parseOffsetBased(context, text, position, position + 3,
3846                                         OffsetIdPrinterParser.INSTANCE_ID_ZERO);
3847             }
3848         }
3849
3850         // parse
3851         PrefixTree tree = getTree(context);
3852         ParsePosition ppos = new ParsePosition(position);
3853         String parsedZoneId = tree.match(text, ppos);
3854         if (parsedZoneId == null) {
3855             if (context.charEquals(nextChar, 'Z')) {
3856                 context.setParsed(ZoneOffset.UTC);
3857                 return position + 1;
3858             }
3859             return ~position;
3860         }
3861         context.setParsed(ZoneId.of(parsedZoneId));
3862         return ppos.getIndex();
3863     }
3864
3865     /**
3866      * Parse an offset following a prefix and set the ZoneId if it is valid.
3867      * To matching the parsing of ZoneId.of the values are not normalized
3868      * to ZoneOffsets.
3869      *
3870      * @param context the parse context

```

```

3865     * @param text the input text
3866     * @param prefixPos start of the prefix
3867     * @param position start of text after the prefix
3868     * @param parser parser for the value after the prefix
3869     * @return the position after the parse
3870     */
3871     private int parseOffsetBased(DateTimeParseContext context, CharSequence text, int prefixPos
        , int position, OffsetIdPrinterParser parser) {
3872         String prefix = text.toString().substring(prefixPos, position).toUpperCase();
3873         if (position >= text.length()) {
3874             context.setParsed(ZoneId.of(prefix));
3875             return position;
3876         }
3877
3878         // 'O' or 'Z' after prefix is not part of a valid ZoneId; use bare prefix
3879         if (text.charAt(position) == 'O' ||
3880             context.charEquals(text.charAt(position), 'Z')) {
3881             context.setParsed(ZoneId.of(prefix));
3882             return position;
3883         }
3884
3885         DateTimeParseContext newContext = context.copy();
3886         int endPos = parser.parse(newContext, text, position);
3887         try {
3888             if (endPos < 0) {
3889                 if (parser == OffsetIdPrinterParser.INSTANCE_ID_Z) {
3890                     return ~prefixPos;
3891                 }
3892                 context.setParsed(ZoneId.of(prefix));
3893                 return position;
3894             }
3895             int offset = (int) newContext.getParsed(OFFSET_SECONDS).longValue();
3896             ZoneOffset zoneOffset = ZoneOffset.ofTotalSeconds(offset);
3897             context.setParsed(ZoneId.ofOffset(prefix, zoneOffset));
3898             return endPos;
3899         } catch (DateTimeException dte) {
3900             return ~prefixPos;
3901         }
3902     }
3903
3904     @Override
3905     public String toString() {
3906         return description;
3907     }
3908 }
3909
3910 //-----
3911 /**
3912  * A String based prefix tree for parsing time-zone names.
3913  */
3914     static class PrefixTree {
3915         protected String key;
3916         protected String value;

```

```

3917     protected char c0;    // performance optimization to avoid the
3918                             // boundary check cost of key.charAt(0)
3919     protected PrefixTree child;
3920     protected PrefixTree sibling;
3921
3922     private PrefixTree(String k, String v, PrefixTree child) {
3923         this.key = k;
3924         this.value = v;
3925         this.child = child;
3926         if (k.length() == 0){
3927             c0 = 0xffff;
3928         } else {
3929             c0 = key.charAt(0);
3930         }
3931     }
3932
3933     /**
3934     * Creates a new prefix parsing tree based on parse context.
3935     *
3936     * @param context the parse context
3937     * @return the tree, not null
3938     */
3939     public static PrefixTree newTree(DateTimeParseContext context) {
3940         //if (!context.isStrict()) {
3941             //    return new LENIENT("", null, null);
3942         //}
3943         if (context.isCaseSensitive()) {
3944             return new PrefixTree("", null, null);
3945         }
3946         return new CI("", null, null);
3947     }
3948
3949     /**
3950     * Creates a new prefix parsing tree.
3951     *
3952     * @param keys a set of strings to build the prefix parsing tree, not null
3953     * @param context the parse context
3954     * @return the tree, not null
3955     */
3956     public static PrefixTree newTree(Set<String> keys, DateTimeParseContext context) {
3957         PrefixTree tree = newTree(context);
3958         for (String k : keys) {
3959             tree.add0(k, k);
3960         }
3961         return tree;
3962     }
3963
3964     /**
3965     * Clone a copy of this tree
3966     */
3967     public PrefixTree copyTree() {
3968         PrefixTree copy = new PrefixTree(key, value, null);
3969         if (child != null) {

```

```

3970         copy.child = child.copyTree();
3971     }
3972     if (sibling != null) {
3973         copy.sibling = sibling.copyTree();
3974     }
3975     return copy;
3976 }
3977
3978
3979 /**
3980  * Adds a pair of {key, value} into the prefix tree.
3981  *
3982  * @param k the key, not null
3983  * @param v the value, not null
3984  * @return true if the pair is added successfully
3985  */
3986 public boolean add(String k, String v) {
3987     return add0(k, v);
3988 }
3989
3990 private boolean add0(String k, String v) {
3991     k = toKey(k);
3992     int prefixLen = prefixLength(k);
3993     if (prefixLen == key.length()) {
3994         if (prefixLen < k.length()) { // down the tree
3995             String subKey = k.substring(prefixLen);
3996             PrefixTree c = child;
3997             while (c != null) {
3998                 if (isEqual(c.c0, subKey.charAt(0))) {
3999                     return c.add0(subKey, v);
4000                 }
4001                 c = c.sibling;
4002             }
4003             // add the node as the child of the current node
4004             c = newNode(subKey, v, null);
4005             c.sibling = child;
4006             child = c;
4007             return true;
4008         }
4009         // have an existing <key, value> already, overwrite it
4010         // if (value != null) {
4011         //     return false;
4012         // }
4013         value = v;
4014         return true;
4015     }
4016     // split the existing node
4017     PrefixTree n1 = newNode(key.substring(prefixLen), value, child);
4018     key = k.substring(0, prefixLen);
4019     child = n1;
4020     if (prefixLen < k.length()) {
4021         PrefixTree n2 = newNode(k.substring(prefixLen), v, null);
4022         child.sibling = n2;

```

```

4023         value = null;
4024     } else {
4025         value = v;
4026     }
4027     return true;
4028 }
4029
4030 /**
4031  * Match text with the prefix tree.
4032  *
4033  * @param text the input text to parse, not null
4034  * @param off the offset position to start parsing at
4035  * @param end the end position to stop parsing
4036  * @return the resulting string, or null if no match found.
4037  */
4038 public String match(CharSequence text, int off, int end) {
4039     if (!prefixOf(text, off, end)){
4040         return null;
4041     }
4042     if (child != null && (off += key.length()) != end) {
4043         PrefixTree c = child;
4044         do {
4045             if (isEqual(c.c0, text.charAt(off))) {
4046                 String found = c.match(text, off, end);
4047                 if (found != null) {
4048                     return found;
4049                 }
4050                 return value;
4051             }
4052             c = c.sibling;
4053         } while (c != null);
4054     }
4055     return value;
4056 }
4057
4058 /**
4059  * Match text with the prefix tree.
4060  *
4061  * @param text the input text to parse, not null
4062  * @param pos the position to start parsing at, from 0 to the text
4063  * length. Upon return, position will be updated to the new parse
4064  * position, or unchanged, if no match found.
4065  * @return the resulting string, or null if no match found.
4066  */
4067 public String match(CharSequence text, ParsePosition pos) {
4068     int off = pos.getIndex();
4069     int end = text.length();
4070     if (!prefixOf(text, off, end)){
4071         return null;
4072     }
4073     off += key.length();
4074     if (child != null && off != end) {
4075         PrefixTree c = child;

```

```

4076         do {
4077             if (isEqual(c.c0, text.charAt(off))) {
4078                 pos.setIndex(off);
4079                 String found = c.match(text, pos);
4080                 if (found != null) {
4081                     return found;
4082                 }
4083                 break;
4084             }
4085             c = c.sibling;
4086         } while (c != null);
4087     }
4088     pos.setIndex(off);
4089     return value;
4090 }
4091
4092 protected String toKey(String k) {
4093     return k;
4094 }
4095
4096 protected PrefixTree newNode(String k, String v, PrefixTree child) {
4097     return new PrefixTree(k, v, child);
4098 }
4099
4100 protected boolean isEqual(char c1, char c2) {
4101     return c1 == c2;
4102 }
4103
4104 protected boolean prefixOf(CharSequence text, int off, int end) {
4105     if (text instanceof String) {
4106         return ((String)text).startsWith(key, off);
4107     }
4108     int len = key.length();
4109     if (len > end - off) {
4110         return false;
4111     }
4112     int off0 = 0;
4113     while (len-- > 0) {
4114         if (!isEqual(key.charAt(off0++), text.charAt(off++))) {
4115             return false;
4116         }
4117     }
4118     return true;
4119 }
4120
4121 private int prefixLength(String k) {
4122     int off = 0;
4123     while (off < k.length() && off < key.length()) {
4124         if (!isEqual(k.charAt(off), key.charAt(off))) {
4125             return off;
4126         }
4127         off++;
4128     }

```



```

4129         return off;
4130     }
4131
4132     /**
4133      * Case Insensitive prefix tree.
4134      */
4135     private static class CI extends PrefixTree {
4136
4137         private CI(String k, String v, PrefixTree child) {
4138             super(k, v, child);
4139         }
4140
4141         @Override
4142         protected CI newNode(String k, String v, PrefixTree child) {
4143             return new CI(k, v, child);
4144         }
4145
4146         @Override
4147         protected boolean isEqual(char c1, char c2) {
4148             return DateTimeParseContext.charEqualsIgnoreCase(c1, c2);
4149         }
4150
4151         @Override
4152         protected boolean prefixOf(CharSequence text, int off, int end) {
4153             int len = key.length();
4154             if (len > end - off) {
4155                 return false;
4156             }
4157             int off0 = 0;
4158             while (len-- > 0) {
4159                 if (!isEqual(key.charAt(off0++), text.charAt(off++))) {
4160                     return false;
4161                 }
4162             }
4163             return true;
4164         }
4165     }
4166
4167     /**
4168      * Lenient prefix tree. Case insensitive and ignores characters
4169      * like space, underscore and slash.
4170      */
4171     private static class LENIENT extends CI {
4172
4173         private LENIENT(String k, String v, PrefixTree child) {
4174             super(k, v, child);
4175         }
4176
4177         @Override
4178         protected CI newNode(String k, String v, PrefixTree child) {
4179             return new LENIENT(k, v, child);
4180         }
4181     }

```

```

4182     private boolean isLenientChar(char c) {
4183         return c == ' ' || c == '_' || c == '/';
4184     }
4185
4186     protected String toKey(String k) {
4187         for (int i = 0; i < k.length(); i++) {
4188             if (isLenientChar(k.charAt(i))) {
4189                 StringBuilder sb = new StringBuilder(k.length());
4190                 sb.append(k, 0, i);
4191                 i++;
4192                 while (i < k.length()) {
4193                     if (!isLenientChar(k.charAt(i))) {
4194                         sb.append(k.charAt(i));
4195                     }
4196                     i++;
4197                 }
4198                 return sb.toString();
4199             }
4200         }
4201         return k;
4202     }
4203
4204     @Override
4205     public String match(CharSequence text, ParsePosition pos) {
4206         int off = pos.getIndex();
4207         int end = text.length();
4208         int len = key.length();
4209         int koff = 0;
4210         while (koff < len && off < end) {
4211             if (isLenientChar(text.charAt(off))) {
4212                 off++;
4213                 continue;
4214             }
4215             if (!isEqual(key.charAt(koff++), text.charAt(off++))) {
4216                 return null;
4217             }
4218         }
4219         if (koff != len) {
4220             return null;
4221         }
4222         if (child != null && off != end) {
4223             int off0 = off;
4224             while (off0 < end && isLenientChar(text.charAt(off0))) {
4225                 off0++;
4226             }
4227             if (off0 < end) {
4228                 PrefixTree c = child;
4229                 do {
4230                     if (isEqual(c.c0, text.charAt(off0))) {
4231                         pos.setIndex(off0);
4232                         String found = c.match(text, pos);
4233                         if (found != null) {
4234                             return found;

```

```

4235         }
4236         break;
4237     }
4238     c = c.sibling;
4239     } while (c != null);
4240 }
4241 }
4242 pos.setIndex(off);
4243 return value;
4244 }
4245 }
4246 }
4247
4248 //-----
4249 /**
4250  * Prints or parses a chronology.
4251  */
4252 static final class ChronoPrinterParser implements DateTimePrinterParser {
4253     /** The text style to output, null means the ID. */
4254     private final TextStyle textStyle;
4255
4256     ChronoPrinterParser(TextStyle textStyle) {
4257         // validated by caller
4258         this.textStyle = textStyle;
4259     }
4260
4261     @Override
4262     public boolean format(DateTimePrintContext context, StringBuilder buf) {
4263         Chronology chrono = context.getValue(TemporalQueries.chronology());
4264         if (chrono == null) {
4265             return false;
4266         }
4267         if (textStyle == null) {
4268             buf.append(chrono.getId());
4269         } else {
4270             buf.append(getChronologyName(chrono, context.getLocale()));
4271         }
4272         return true;
4273     }
4274
4275     @Override
4276     public int parse(DateTimeParseContext context, CharSequence text, int position) {
4277         // simple looping parser to find the chronology
4278         if (position < 0 || position > text.length()) {
4279             throw new IndexOutOfBoundsException();
4280         }
4281         Set<Chronology> chronos = Chronology.getAvailableChronologies();
4282         Chronology bestMatch = null;
4283         int matchLen = -1;
4284         for (Chronology chrono : chronos) {
4285             String name;
4286             if (textStyle == null) {
4287                 name = chrono.getId();

```

```

4288         } else {
4289             name = getChronologyName(chrono, context.getLocale());
4290         }
4291         int nameLen = name.length();
4292         if (nameLen > matchLen && context.subSequenceEquals(text, position, name, 0,
4293             nameLen)) {
4294             bestMatch = chrono;
4295             matchLen = nameLen;
4296         }
4297         if (bestMatch == null) {
4298             return ~position;
4299         }
4300         context.setParsed(bestMatch);
4301         return position + matchLen;
4302     }
4303
4304     /**
4305      * Returns the chronology name of the given chrono in the given locale
4306      * if available, or the chronology Id otherwise. The regular ResourceBundle
4307      * search path is used for looking up the chronology name.
4308      *
4309      * @param chrono the chronology, not null
4310      * @param locale the locale, not null
4311      * @return the chronology name of chrono in locale, or the id if no name is available
4312      * @throws NullPointerException if chrono or locale is null
4313      */
4314     private String getChronologyName(Chronology chrono, Locale locale) {
4315         String key = "calendarname." + chrono.getCalendarType();
4316         String name = DateTimeTextProvider.getLocalizedResource(key, locale);
4317         return name != null ? name : chrono.getId();
4318     }
4319 }
4320
4321 //-----
4322 /**
4323  * Prints or parses a localized pattern.
4324  */
4325 static final class LocalizedPrinterParser implements DateTimePrinterParser {
4326     /** Cache of formatters. */
4327     private static final ConcurrentMap<String, DateTimeFormatter> FORMATTER_CACHE = new
4328         ConcurrentHashMap<>(16, 0.75f, 2);
4329
4330     private final FormatStyle dateStyle;
4331     private final FormatStyle timeStyle;
4332
4333     /**
4334      * Constructor.
4335      *
4336      * @param dateStyle the date style to use, may be null
4337      * @param timeStyle the time style to use, may be null
4338      */
4339     LocalizedPrinterParser(FormatStyle dateStyle, FormatStyle timeStyle) {

```

```

4339         // validated by caller
4340         this.dateStyle = dateStyle;
4341         this.timeStyle = timeStyle;
4342     }
4343
4344     @Override
4345     public boolean format(DateTimePrintContext context, StringBuilder buf) {
4346         Chronology chrono = Chronology.from(context.getTemporal());
4347         return formatter(context.getLocale(), chrono).toPrinterParser(false).format(context,
4348             buf);
4349     }
4350
4351     @Override
4352     public int parse(DateTimeParseContext context, CharSequence text, int position) {
4353         Chronology chrono = context.getEffectiveChronology();
4354         return formatter(context.getLocale(), chrono).toPrinterParser(false).parse(context,
4355             text, position);
4356     }
4357
4358     /**
4359      * Gets the formatter to use.
4360      * <p>
4361      * The formatter will be the most appropriate to use for the date and time style in the
4362      * locale.
4363      * For example, some locales will use the month name while others will use the number.
4364      *
4365      * @param locale the locale to use, not null
4366      * @param chrono the chronology to use, not null
4367      * @return the formatter, not null
4368      * @throws IllegalArgumentException if the formatter cannot be found
4369      */
4370     private DateTimeFormatter formatter(Locale locale, Chronology chrono) {
4371         String key = chrono.getId() + '|' + locale.toString() + '|' + dateStyle + timeStyle;
4372         DateTimeFormatter formatter = FORMATTER_CACHE.get(key);
4373         if (formatter == null) {
4374             String pattern = getLocalizedDateTimePattern(dateStyle, timeStyle, chrono, locale);
4375             formatter = new DateTimeFormatterBuilder().appendPattern(pattern).toFormatter(
4376                 locale);
4377             DateTimeFormatter old = FORMATTER_CACHE.putIfAbsent(key, formatter);
4378             if (old != null) {
4379                 formatter = old;
4380             }
4381         }
4382         return formatter;
4383     }
4384
4385     @Override
4386     public String toString() {
4387         return "Localized(" + (dateStyle != null ? dateStyle : "") + "," +
4388             (timeStyle != null ? timeStyle : "") + ")";
4389     }
4390 }

```

```

4388 //-----
4389 /**
4390  * Prints or parses a localized pattern from a localized field.
4391  * The specific formatter and parameters is not selected until the
4392  * the field is to be printed or parsed.
4393  * The locale is needed to select the proper WeekFields from which
4394  * the field for day-of-week, week-of-month, or week-of-year is selected.
4395  */
4396 static final class WeekBasedFieldPrinterParser implements DateTimePrinterParser {
4397     private char chr;
4398     private int count;
4399
4400     /**
4401      * Constructor.
4402      *
4403      * @param chr the pattern format letter that added this PrinterParser.
4404      * @param count the repeat count of the format letter
4405      */
4406     WeekBasedFieldPrinterParser(char chr, int count) {
4407         this.chr = chr;
4408         this.count = count;
4409     }
4410
4411     @Override
4412     public boolean format(DateTimePrintContext context, StringBuilder buf) {
4413         return printerParser(context.getLocale()).format(context, buf);
4414     }
4415
4416     @Override
4417     public int parse(DateTimeParseContext context, CharSequence text, int position) {
4418         return printerParser(context.getLocale()).parse(context, text, position);
4419     }
4420
4421     /**
4422      * Gets the printerParser to use based on the field and the locale.
4423      *
4424      * @param locale the locale to use, not null
4425      * @return the formatter, not null
4426      * @throws IllegalArgumentException if the formatter cannot be found
4427      */
4428     private DateTimePrinterParser printerParser(Locale locale) {
4429         WeekFields weekDef = WeekFields.of(locale);
4430         TemporalField field = null;
4431         switch (chr) {
4432             case 'Y':
4433                 field = weekDef.weekBasedYear();
4434                 if (count == 2) {
4435                     return new ReducedPrinterParser(field, 2, 2, 0, ReducedPrinterParser.
4436                         BASE_DATE, 0);
4437                 } else {
4438                     return new NumberPrinterParser(field, count, 19,
4439                         (count < 4) ? SignStyle.NORMAL : SignStyle.EXCEEDS_PAD, -1);
4440                 }

```

```

4440         case 'e':
4441         case 'c':
4442             field = weekDef.dayOfWeek();
4443             break;
4444         case 'w':
4445             field = weekDef.weekOfWeekBasedYear();
4446             break;
4447         case 'W':
4448             field = weekDef.weekOfMonth();
4449             break;
4450         default:
4451             throw new IllegalStateException("unreachable");
4452     }
4453     return new NumberPrinterParser(field, (count == 2 ? 2 : 1), 2, SignStyle.NOT_NEGATIVE);
4454 }
4455
4456 @Override
4457 public String toString() {
4458     StringBuilder sb = new StringBuilder(30);
4459     sb.append("Localized(");
4460     if (chr == 'Y') {
4461         if (count == 1) {
4462             sb.append("WeekBasedYear");
4463         } else if (count == 2) {
4464             sb.append("ReducedValue(WeekBasedYear,2,2,2000-01-01)");
4465         } else {
4466             sb.append("WeekBasedYear,").append(count).append(",")
4467                 .append(19).append(",")
4468                 .append((count < 4) ? SignStyle.NORMAL : SignStyle.EXCEEDS_PAD);
4469         }
4470     } else {
4471         switch (chr) {
4472             case 'c':
4473             case 'e':
4474                 sb.append("DayOfWeek");
4475                 break;
4476             case 'w':
4477                 sb.append("WeekOfWeekBasedYear");
4478                 break;
4479             case 'W':
4480                 sb.append("WeekOfMonth");
4481                 break;
4482             default:
4483                 break;
4484         }
4485         sb.append(",");
4486         sb.append(count);
4487     }
4488     sb.append(")");
4489     return sb.toString();
4490 }
4491 }
4492

```

```

4493 //-----
4494 /**
4495  * Length comparator.
4496  */
4497 static final Comparator<String> LENGTH_SORT = new Comparator<String>() {
4498     @Override
4499     public int compare(String str1, String str2) {
4500         return str1.length() == str2.length() ? str1.compareTo(str2) : str1.length() - str2.
            length();
4501     }
4502 };
4503 }

```

6.3 DateTimeParseContext.java

Secuencia 100: java/time/format/DateTimeParseContext.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2008-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:

```



```
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
62 package java.time.format;
63
64 import java.time.ZoneId;
65 import java.time.chrono.Chronology;
66 import java.time.chrono.IsoChronology;
67 import java.time.temporal.TemporalAccessor;
68 import java.time.temporal.TemporalField;
69 import java.util.ArrayList;
70 import java.util.Locale;
71 import java.util.Objects;
72 import java.util.Set;
73 import java.util.function.Consumer;
74
75 /**
76  * Context object used during date and time parsing.
77  * <p>
78  * This class represents the current state of the parse.
79  * It has the ability to store and retrieve the parsed values and manage optional segments.
80  * It also provides key information to the parsing methods.
81  * <p>
82  * Once parsing is complete, the {@link #toUnresolved()} is used to obtain the unresolved
83  * result data. The {@link #toResolved()} is used to obtain the resolved result.
84  *
85  * @implSpec
86  * This class is a mutable context intended for use from a single thread.
87  * Usage of the class is thread-safe within standard parsing as a new instance of this class
88  * is automatically created for each parse and parsing is single-threaded
89  *
90  * @since 1.8
```

```

91  */
92  final class DateTimeParseContext {
93
94      /**
95       * The formatter, not null.
96       */
97      private DateTimeFormatter formatter;
98      /**
99       * Whether to parse using case sensitively.
100     */
101     private boolean caseSensitive = true;
102     /**
103      * Whether to parse using strict rules.
104      */
105     private boolean strict = true;
106     /**
107      * The list of parsed data.
108      */
109     private final ArrayList<Parsed> parsed = new ArrayList<>();
110     /**
111      * List of Consumers<Chronology> to be notified if the Chronology changes.
112      */
113     private ArrayList<Consumer<Chronology>> chronoListeners = null;
114
115     /**
116      * Creates a new instance of the context.
117      *
118      * @param formatter the formatter controlling the parse, not null
119      */
120     DateTimeParseContext(DateTimeFormatter formatter) {
121         super();
122         this.formatter = formatter;
123         parsed.add(new Parsed());
124     }
125
126     /**
127      * Creates a copy of this context.
128      * This retains the case sensitive and strict flags.
129      */
130     DateTimeParseContext copy() {
131         DateTimeParseContext newContext = new DateTimeParseContext(formatter);
132         newContext.caseSensitive = caseSensitive;
133         newContext.strict = strict;
134         return newContext;
135     }
136
137     //-----
138     /**
139      * Gets the locale.
140      * <p>
141      * This locale is used to control localization in the parse except
142      * where localization is controlled by the DecimalStyle.
143      *

```

```
144     * @return the locale, not null
145     */
146     Locale getLocale() {
147         return formatter.getLocale();
148     }
149
150     /**
151     * Gets the DecimalStyle.
152     * <p>
153     * The DecimalStyle controls the numeric parsing.
154     *
155     * @return the DecimalStyle, not null
156     */
157     DecimalStyle getDecimalStyle() {
158         return formatter.getDecimalStyle();
159     }
160
161     /**
162     * Gets the effective chronology during parsing.
163     *
164     * @return the effective parsing chronology, not null
165     */
166     Chronology getEffectiveChronology() {
167         Chronology chrono = currentParsed().chrono;
168         if (chrono == null) {
169             chrono = formatter.getChronology();
170             if (chrono == null) {
171                 chrono = IsoChronology.INSTANCE;
172             }
173         }
174         return chrono;
175     }
176
177     //-----
178     /**
179     * Checks if parsing is case sensitive.
180     *
181     * @return true if parsing is case sensitive, false if case insensitive
182     */
183     boolean isCaseSensitive() {
184         return caseSensitive;
185     }
186
187     /**
188     * Sets whether the parsing is case sensitive or not.
189     *
190     * @param caseSensitive changes the parsing to be case sensitive or not from now on
191     */
192     void setCaseSensitive(boolean caseSensitive) {
193         this.caseSensitive = caseSensitive;
194     }
195
196     //-----
```

```

197  /**
198   * Helper to compare two {@code CharSequence} instances.
199   * This uses {@link #isCaseSensitive()}.
200   *
201   * @param cs1 the first character sequence, not null
202   * @param offset1 the offset into the first sequence, valid
203   * @param cs2 the second character sequence, not null
204   * @param offset2 the offset into the second sequence, valid
205   * @param length the length to check, valid
206   * @return true if equal
207   */
208  boolean subSequenceEquals(CharSequence cs1, int offset1, CharSequence cs2, int offset2, int
    length) {
209      if (offset1 + length > cs1.length() || offset2 + length > cs2.length()) {
210          return false;
211      }
212      if (isCaseSensitive()) {
213          for (int i = 0; i < length; i++) {
214              char ch1 = cs1.charAt(offset1 + i);
215              char ch2 = cs2.charAt(offset2 + i);
216              if (ch1 != ch2) {
217                  return false;
218              }
219          }
220      } else {
221          for (int i = 0; i < length; i++) {
222              char ch1 = cs1.charAt(offset1 + i);
223              char ch2 = cs2.charAt(offset2 + i);
224              if (ch1 != ch2 && Character.toUpperCase(ch1) != Character.toUpperCase(ch2) &&
225                  Character.toLowerCase(ch1) != Character.toLowerCase(ch2)) {
226                  return false;
227              }
228          }
229      }
230      return true;
231  }
232
233  /**
234   * Helper to compare two {@code char}.
235   * This uses {@link #isCaseSensitive()}.
236   *
237   * @param ch1 the first character
238   * @param ch2 the second character
239   * @return true if equal
240   */
241  boolean charEquals(char ch1, char ch2) {
242      if (isCaseSensitive()) {
243          return ch1 == ch2;
244      }
245      return charEqualsIgnoreCase(ch1, ch2);
246  }
247
248  /**

```

```
249     * Compares two characters ignoring case.
250     *
251     * @param c1  the first
252     * @param c2  the second
253     * @return true if equal
254     */
255     static boolean charEqualsIgnoreCase(char c1, char c2) {
256         return c1 == c2 ||
257             Character.toUpperCase(c1) == Character.toUpperCase(c2) ||
258             Character.toLowerCase(c1) == Character.toLowerCase(c2);
259     }
260
261     //-----
262     /**
263     * Checks if parsing is strict.
264     * <p>
265     * Strict parsing requires exact matching of the text and sign styles.
266     *
267     * @return true if parsing is strict, false if lenient
268     */
269     boolean isStrict() {
270         return strict;
271     }
272
273     /**
274     * Sets whether parsing is strict or lenient.
275     *
276     * @param strict  changes the parsing to be strict or lenient from now on
277     */
278     void setStrict(boolean strict) {
279         this.strict = strict;
280     }
281
282     //-----
283     /**
284     * Starts the parsing of an optional segment of the input.
285     */
286     void startOptional() {
287         parsed.add(currentParsed().copy());
288     }
289
290     /**
291     * Ends the parsing of an optional segment of the input.
292     *
293     * @param successful  whether the optional segment was successfully parsed
294     */
295     void endOptional(boolean successful) {
296         if (successful) {
297             parsed.remove(parsed.size() - 2);
298         } else {
299             parsed.remove(parsed.size() - 1);
300         }
301     }
```

```

302
303 //-----
304 /**
305  * Gets the currently active temporal objects.
306  *
307  * @return the current temporal objects, not null
308  */
309 private Parsed currentParsed() {
310     return parsed.get(parsed.size() - 1);
311 }
312
313 /**
314  * Gets the unresolved result of the parse.
315  *
316  * @return the result of the parse, not null
317  */
318 Parsed toUnresolved() {
319     return currentParsed();
320 }
321
322 /**
323  * Gets the resolved result of the parse.
324  *
325  * @return the result of the parse, not null
326  */
327 TemporalAccessor toResolved(ResolverStyle resolverStyle, Set<TemporalField> resolverFields) {
328     Parsed parsed = currentParsed();
329     parsed.chrono = getEffectiveChronology();
330     parsed.zone = (parsed.zone != null ? parsed.zone : formatter.getZone());
331     return parsed.resolve(resolverStyle, resolverFields);
332 }
333
334 //-----
335 /**
336  * Gets the first value that was parsed for the specified field.
337  * <p>
338  * This searches the results of the parse, returning the first value found
339  * for the specified field. No attempt is made to derive a value.
340  * The field may have an out of range value.
341  * For example, the day-of-month might be set to 50, or the hour to 1000.
342  *
343  * @param field the field to query from the map, null returns null
344  * @return the value mapped to the specified field, null if field was not parsed
345  */
346 Long getParsed(TemporalField field) {
347     return currentParsed().fieldValues.get(field);
348 }
349
350 /**
351  * Stores the parsed field.
352  * <p>
353  * This stores a field-value pair that has been parsed.

```

```

355     * The value stored may be out of range for the field - no checks are performed.
356     *
357     * @param field the field to set in the field-value map, not null
358     * @param value the value to set in the field-value map
359     * @param errorPos the position of the field being parsed
360     * @param successPos the position after the field being parsed
361     * @return the new position
362     */
363     int setParsedField(TemporalField field, long value, int errorPos, int successPos) {
364         Objects.requireNonNull(field, "field");
365         Long old = currentParsed().fieldValues.put(field, value);
366         return (old != null && old.longValue() != value) ? ~errorPos : successPos;
367     }
368
369     /**
370     * Stores the parsed chronology.
371     * <p>
372     * This stores the chronology that has been parsed.
373     * No validation is performed other than ensuring it is not null.
374     * <p>
375     * The list of listeners is copied and cleared so that each
376     * listener is called only once. A listener can add itself again
377     * if it needs to be notified of future changes.
378     *
379     * @param chrono the parsed chronology, not null
380     */
381     void setParsed(Chronology chrono) {
382         Objects.requireNonNull(chrono, "chrono");
383         currentParsed().chrono = chrono;
384         if (chronoListeners != null && !chronoListeners.isEmpty()) {
385             @SuppressWarnings({"rawtypes", "unchecked"})
386             Consumer<Chronology>[] tmp = new Consumer[1];
387             Consumer<Chronology>[] listeners = chronoListeners.toArray(tmp);
388             chronoListeners.clear();
389             for (Consumer<Chronology> l : listeners) {
390                 l.accept(chrono);
391             }
392         }
393     }
394
395     /**
396     * Adds a Consumer<Chronology> to the list of listeners to be notified
397     * if the Chronology changes.
398     * @param listener a Consumer<Chronology> to be called when Chronology changes
399     */
400     void addChronoChangedListener(Consumer<Chronology> listener) {
401         if (chronoListeners == null) {
402             chronoListeners = new ArrayList<Consumer<Chronology>>();
403         }
404         chronoListeners.add(listener);
405     }
406
407     /**

```

```

408     * Stores the parsed zone.
409     * <p>
410     * This stores the zone that has been parsed.
411     * No validation is performed other than ensuring it is not null.
412     *
413     * @param zone the parsed zone, not null
414     */
415     void setParsed(ZoneId zone) {
416         Objects.requireNonNull(zone, "zone");
417         currentParsed().zone = zone;
418     }
419
420     /**
421     * Stores the parsed leap second.
422     */
423     void setParsedLeapSecond() {
424         currentParsed().leapSecond = true;
425     }
426
427     //-----
428     /**
429     * Returns a string version of the context for debugging.
430     *
431     * @return a string representation of the context data, not null
432     */
433     @Override
434     public String toString() {
435         return currentParsed().toString();
436     }
437
438 }

```

6.4 DateTimeParseException.java

Secuencia 101: java/time/format/DateTimeParseException.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *

```



```
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2008-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.format;
63
64  import java.time.DateTimeException;
65
66  /**
67   * An exception thrown when an error occurs during parsing.
68   * <p>
69   * This exception includes the text being parsed and the error index.
70   *
71   * @implSpec
```

```

72  * This class is intended for use in a single thread.
73  *
74  * @since 1.8
75  */
76  public class DateTimeParseException extends DateTimeException {
77
78      /**
79       * Serialization version.
80       */
81      private static final long serialVersionUID = 4304633501674722597L;
82
83      /**
84       * The text that was being parsed.
85       */
86      private final String parsedString;
87      /**
88       * The error index in the text.
89       */
90      private final int errorIndex;
91
92      /**
93       * Constructs a new exception with the specified message.
94       *
95       * @param message the message to use for this exception, may be null
96       * @param parsedData the parsed text, should not be null
97       * @param errorIndex the index in the parsed string that was invalid, should be a valid index
98       */
99      public DateTimeParseException(String message, CharSequence parsedData, int errorIndex) {
100          super(message);
101          this.parsedString = parsedData.toString();
102          this.errorIndex = errorIndex;
103      }
104
105      /**
106       * Constructs a new exception with the specified message and cause.
107       *
108       * @param message the message to use for this exception, may be null
109       * @param parsedData the parsed text, should not be null
110       * @param errorIndex the index in the parsed string that was invalid, should be a valid index
111       * @param cause the cause exception, may be null
112       */
113      public DateTimeParseException(String message, CharSequence parsedData, int errorIndex,
114          Throwable cause) {
115          super(message, cause);
116          this.parsedString = parsedData.toString();
117          this.errorIndex = errorIndex;
118      }
119
120      /**
121       * Returns the string that was being parsed.
122       *
123       * @return the string that was being parsed, should not be null.

```

```
124     */
125     public String getParsedString() {
126         return parsedString;
127     }
128
129     /**
130      * Returns the index where the error was found.
131      *
132      * @return the index in the parsed string that was invalid, should be a valid index
133      */
134     public int getErrorIndex() {
135         return errorIndex;
136     }
137
138 }
```

6.5 DateTimePrintContext.java

Secuencia 102: java/time/format/DateTimePrintContext.java

```
1  /*
2   * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3   * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4   *
5   *
6   *
7   *
8   *
9   *
10  *
11  *
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27   *
28   *
29   *
30   *
31   *
32   * Copyright (c) 2011-2012, Stephen Colebourne & Michael Nascimento Santos
33   *
34   * All rights reserved.
```

```
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.format;
63
64  import static java.time.temporal.ChronoField.EPOCH_DAY;
65  import static java.time.temporal.ChronoField.INSTANT_SECONDS;
66  import static java.time.temporal.ChronoField.OFFSET_SECONDS;
67
68  import java.time.DateTimeException;
69  import java.time.Instant;
70  import java.time.ZoneId;
71  import java.time.ZoneOffset;
72  import java.time.chrono.ChronoLocalDate;
73  import java.time.chrono.Chronology;
74  import java.time.chrono.IsoChronology;
75  import java.time.temporal.ChronoField;
76  import java.time.temporal.TemporalAccessor;
77  import java.time.temporal.TemporalField;
78  import java.time.temporal.TemporalQueries;
79  import java.time.temporal.TemporalQuery;
80  import java.time.temporal.ValueRange;
81  import java.util.Locale;
82  import java.util.Objects;
83
84  /**
85   * Context object used during date and time printing.
86   * <p>
87   * This class provides a single wrapper to items used in the format.
```

```

88  *
89  * @implSpec
90  * This class is a mutable context intended for use from a single thread.
91  * Usage of the class is thread-safe within standard printing as the framework creates
92  * a new instance of the class for each format and printing is single-threaded.
93  *
94  * @since 1.8
95  */
96  final class DateTimePrintContext {
97
98      /**
99       * The temporal being output.
100     */
101     private TemporalAccessor temporal;
102     /**
103      * The formatter, not null.
104     */
105     private DateTimeFormatter formatter;
106     /**
107      * Whether the current formatter is optional.
108     */
109     private int optional;
110
111     /**
112      * Creates a new instance of the context.
113      *
114      * @param temporal the temporal object being output, not null
115      * @param formatter the formatter controlling the format, not null
116     */
117     DateTimePrintContext(TemporalAccessor temporal, DateTimeFormatter formatter) {
118         super();
119         this.temporal = adjust(temporal, formatter);
120         this.formatter = formatter;
121     }
122
123     private static TemporalAccessor adjust(final TemporalAccessor temporal, DateTimeFormatter
        formatter) {
124         // normal case first (early return is an optimization)
125         Chronology overrideChrono = formatter.getChronology();
126         ZoneId overrideZone = formatter.getZone();
127         if (overrideChrono == null && overrideZone == null) {
128             return temporal;
129         }
130
131         // ensure minimal change (early return is an optimization)
132         Chronology temporalChrono = temporal.query(TemporalQueries.chronology());
133         ZoneId temporalZone = temporal.query(TemporalQueries.zoneId());
134         if (Objects.equals(overrideChrono, temporalChrono)) {
135             overrideChrono = null;
136         }
137         if (Objects.equals(overrideZone, temporalZone)) {
138             overrideZone = null;
139         }

```

```

140     if (overrideChrono == null && overrideZone == null) {
141         return temporal;
142     }
143
144     // make adjustment
145     final Chronology effectiveChrono = (overrideChrono != null ? overrideChrono :
        temporalChrono);
146     if (overrideZone != null) {
147         // if have zone and instant, calculation is simple, defaulting chrono if necessary
148         if (temporal.isSupported(INSTANT_SECONDS)) {
149             Chronology chrono = (effectiveChrono != null ? effectiveChrono : IsoChronology.
                INSTANCE);
150             return chrono.zonedDateTime(Instant.from(temporal), overrideZone);
151         }
152         // block changing zone on OffsetTime, and similar problem cases
153         if (overrideZone.normalized() instanceof ZoneOffset && temporal.isSupported(
            OFFSET_SECONDS) &&
154             temporal.get(OFFSET_SECONDS) != overrideZone.getRules().getOffset(Instant.EPOCH
                ).getTotalSeconds()) {
155             throw new DateTimeException("Unable to apply override zone '" + overrideZone +
156                 "' because the temporal object being formatted has a different offset but"
157                 +
158                 " does not represent an instant: " + temporal);
159         }
160     }
161     final ZoneId effectiveZone = (overrideZone != null ? overrideZone : temporalZone);
162     final ChronoLocalDate effectiveDate;
163     if (overrideChrono != null) {
164         if (temporal.isSupported(EPOCH_DAY)) {
165             effectiveDate = effectiveChrono.date(temporal);
166         } else {
167             // check for date fields other than epoch-day, ignoring case of converting null to
168             // ISO
169             if (!(overrideChrono == IsoChronology.INSTANCE && temporalChrono == null)) {
170                 for (ChronoField f : ChronoField.values()) {
171                     if (f.isDateBased() && temporal.isSupported(f)) {
172                         throw new DateTimeException("Unable to apply override chronology '" +
173                             overrideChrono +
174                             "' because the temporal object being formatted contains date
175                                 fields but" +
176                                 " does not represent a whole date: " + temporal);
177                     }
178                 }
179             }
180             effectiveDate = null;
181         }
182     } else {
183         effectiveDate = null;
184     }
185
186     // combine available data
187     // this is a non-standard temporal that is almost a pure delegate
188     // this better handles map-like underlying temporal instances

```

```

185     return new TemporalAccessor() {
186         @Override
187         public boolean isSupported(TemporalField field) {
188             if (effectiveDate != null && field.isDateBased()) {
189                 return effectiveDate.isSupported(field);
190             }
191             return temporal.isSupported(field);
192         }
193         @Override
194         public ValueRange range(TemporalField field) {
195             if (effectiveDate != null && field.isDateBased()) {
196                 return effectiveDate.range(field);
197             }
198             return temporal.range(field);
199         }
200         @Override
201         public long getLong(TemporalField field) {
202             if (effectiveDate != null && field.isDateBased()) {
203                 return effectiveDate.getLong(field);
204             }
205             return temporal.getLong(field);
206         }
207         @SuppressWarnings("unchecked")
208         @Override
209         public <R> R query(TemporalQuery<R> query) {
210             if (query == TemporalQueries.chronology()) {
211                 return (R) effectiveChrono;
212             }
213             if (query == TemporalQueries.zoneId()) {
214                 return (R) effectiveZone;
215             }
216             if (query == TemporalQueries.precision()) {
217                 return temporal.query(query);
218             }
219             return query.queryFrom(this);
220         }
221     };
222 }
223
224 //-----
225 /**
226  * Gets the temporal object being output.
227  *
228  * @return the temporal object, not null
229  */
230 TemporalAccessor getTemporal() {
231     return temporal;
232 }
233
234 /**
235  * Gets the locale.
236  * <p>
237  * This locale is used to control localization in the format output except

```

```

238     * where localization is controlled by the DecimalStyle.
239     *
240     * @return the locale, not null
241     */
242     Locale getLocale() {
243         return formatter.getLocale();
244     }
245
246     /**
247     * Gets the DecimalStyle.
248     * <p>
249     * The DecimalStyle controls the localization of numeric output.
250     *
251     * @return the DecimalStyle, not null
252     */
253     DecimalStyle getDecimalStyle() {
254         return formatter.getDecimalStyle();
255     }
256
257     //-----
258     /**
259     * Starts the printing of an optional segment of the input.
260     */
261     void startOptional() {
262         this.optional++;
263     }
264
265     /**
266     * Ends the printing of an optional segment of the input.
267     */
268     void endOptional() {
269         this.optional--;
270     }
271
272     /**
273     * Gets a value using a query.
274     *
275     * @param query the query to use, not null
276     * @return the result, null if not found and optional is true
277     * @throws DateTimeException if the type is not available and the section is not optional
278     */
279     <R> R getValue(TemporalQuery<R> query) {
280         R result = temporal.query(query);
281         if (result == null && optional == 0) {
282             throw new DateTimeException("Unable to extract value: " + temporal.getClass());
283         }
284         return result;
285     }
286
287     /**
288     * Gets the value of the specified field.
289     * <p>
290     * This will return the value for the specified field.

```



```

291     *
292     * @param field the field to find, not null
293     * @return the value, null if not found and optional is true
294     * @throws DateTimeException if the field is not available and the section is not optional
295     */
296     Long getValue(TemporalField field) {
297         try {
298             return temporal.getLong(field);
299         } catch (DateTimeException ex) {
300             if (optional > 0) {
301                 return null;
302             }
303             throw ex;
304         }
305     }
306
307     //-----
308     /**
309     * Returns a string version of the context for debugging.
310     *
311     * @return a string representation of the context, not null
312     */
313     @Override
314     public String toString() {
315         return temporal.toString();
316     }
317
318 }

```

6.6 DateTimeTextProvider.java

Secuencia 103: java/time/format/DateTimeTextProvider.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *

```

```
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2011-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.format;
63
64  import static java.time.temporal.ChronoField.AMPM_OF_DAY;
65  import static java.time.temporal.ChronoField.DAY_OF_WEEK;
66  import static java.time.temporal.ChronoField.ERA;
67  import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
68
69  import java.time.chrono.Chronology;
70  import java.time.chrono.IsoChronology;
71  import java.time.chrono.JapaneseChronology;
72  import java.time.temporal.ChronoField;
73  import java.time.temporal.IsoFields;
74  import java.time.temporal.TemporalField;
```

```
75 import java.util.AbstractMap.SimpleImmutableEntry;
76 import java.util.ArrayList;
77 import java.util.Calendar;
78 import java.util.Collections;
79 import java.util.Comparator;
80 import java.util.HashMap;
81 import java.util.Iterator;
82 import java.util.List;
83 import java.util.Locale;
84 import java.util.Map;
85 import java.util.Map.Entry;
86 import java.util.ResourceBundle;
87 import java.util.concurrent.ConcurrentHashMap;
88 import java.util.concurrent.ConcurrentMap;
89
90 import sun.util.locale.provider.CalendarDataUtility;
91 import sun.util.locale.provider.LocaleProviderAdapter;
92 import sun.util.locale.provider.LocaleResources;
93
94 /**
95  * A provider to obtain the textual form of a date-time field.
96  *
97  * @implSpec
98  * Implementations must be thread-safe.
99  * Implementations should cache the textual information.
100  *
101  * @since 1.8
102  */
103 class DateTimeTextProvider {
104
105     /** Cache. */
106     private static final ConcurrentMap<Entry<TemporalField, Locale>, Object> CACHE = new
        ConcurrentHashMap<>(16, 0.75f, 2);
107
108     /** Comparator. */
109     private static final Comparator<Entry<String, Long>> COMPARATOR = new Comparator<Entry<String,
        Long>>() {
110         @Override
111         public int compare(Entry<String, Long> obj1, Entry<String, Long> obj2) {
112             return obj2.getKey().length() - obj1.getKey().length(); // longest to shortest
113         }
114     };
115
116     DateTimeTextProvider() {}
117
118     /**
119     * Gets the provider of text.
120     *
121     * @return the provider, not null
122     */
123     static DateTimeTextProvider getInstance() {
124         return new DateTimeTextProvider();
125     }
```

```

126  /**
127   * Gets the text for the specified field, locale and style
128   * for the purpose of formatting.
129   * <p>
130   * The text associated with the value is returned.
131   * The null return value should be used if there is no applicable text, or
132   * if the text would be a numeric representation of the value.
133   *
134   * @param field  the field to get text for, not null
135   * @param value  the field value to get text for, not null
136   * @param style  the style to get text for, not null
137   * @param locale the locale to get text for, not null
138   * @return the text for the field value, null if no text found
139   */
140  public String getText(TemporalField field, long value, TextStyle style, Locale locale) {
141      Object store = findStore(field, locale);
142      if (store instanceof LocaleStore) {
143          return ((LocaleStore) store).getText(value, style);
144      }
145      return null;
146  }
147
148  /**
149   * Gets the text for the specified chrono, field, locale and style
150   * for the purpose of formatting.
151   * <p>
152   * The text associated with the value is returned.
153   * The null return value should be used if there is no applicable text, or
154   * if the text would be a numeric representation of the value.
155   *
156   * @param chrono the Chronology to get text for, not null
157   * @param field  the field to get text for, not null
158   * @param value  the field value to get text for, not null
159   * @param style  the style to get text for, not null
160   * @param locale the locale to get text for, not null
161   * @return the text for the field value, null if no text found
162   */
163  public String getText(Chronology chrono, TemporalField field, long value,
164                      TextStyle style, Locale locale) {
165      if (chrono == IsoChronology.INSTANCE
166          || !(field instanceof ChronoField)) {
167          return getText(field, value, style, locale);
168      }
169
170      int fieldIndex;
171      int fieldValue;
172      if (field == ERA) {
173          fieldIndex = Calendar.ERA;
174          if (chrono == JapaneseChronology.INSTANCE) {
175              if (value == -999) {
176                  fieldValue = 0;
177              } else {
178                  fieldValue = (int) value + 2;

```

```

179         }
180     } else {
181         fieldValue = (int) value;
182     }
183 } else if (field == MONTH_OF_YEAR) {
184     fieldIndex = Calendar.MONTH;
185     fieldValue = (int) value - 1;
186 } else if (field == DAY_OF_WEEK) {
187     fieldIndex = Calendar.DAY_OF_WEEK;
188     fieldValue = (int) value + 1;
189     if (fieldValue > 7) {
190         fieldValue = Calendar.SUNDAY;
191     }
192 } else if (field == AMPM_OF_DAY) {
193     fieldIndex = Calendar.AM_PM;
194     fieldValue = (int) value;
195 } else {
196     return null;
197 }
198 return CalendarDataUtility.retrieveJavaTimeFieldValueName(
199     chrono.getCalendarType(), fieldIndex, fieldValue, style.toCalendarStyle(), locale);
200 }
201
202 /**
203  * Gets an iterator of text to field for the specified field, locale and style
204  * for the purpose of parsing.
205  * <p>
206  * The iterator must be returned in order from the longest text to the shortest.
207  * <p>
208  * The null return value should be used if there is no applicable parsable text, or
209  * if the text would be a numeric representation of the value.
210  * Text can only be parsed if all the values for that field-style-locale combination are unique
211  *
212  * @param field the field to get text for, not null
213  * @param style the style to get text for, null for all parsable text
214  * @param locale the locale to get text for, not null
215  * @return the iterator of text to field pairs, in order from longest text to shortest text,
216  *         null if the field or style is not parsable
217  */
218 public Iterator<Entry<String, Long>> getTextIterator(TemporalField field, TextStyle style,
219     Locale locale) {
220     Object store = findStore(field, locale);
221     if (store instanceof LocaleStore) {
222         return ((LocaleStore) store).getTextIterator(style);
223     }
224     return null;
225 }
226
227 /**
228  * Gets an iterator of text to field for the specified chrono, field, locale and style
229  * for the purpose of parsing.
230  * <p>

```

```

230     * The iterator must be returned in order from the longest text to the shortest.
231     * <p>
232     * The null return value should be used if there is no applicable parsable text, or
233     * if the text would be a numeric representation of the value.
234     * Text can only be parsed if all the values for that field-style-locale combination are unique
235     *
236     * @param chrono the Chronology to get text for, not null
237     * @param field the field to get text for, not null
238     * @param style the style to get text for, null for all parsable text
239     * @param locale the locale to get text for, not null
240     * @return the iterator of text to field pairs, in order from longest text to shortest text,
241     *         null if the field or style is not parsable
242     */
243     public Iterator<Entry<String, Long>> getTextIterator(Chronology chrono, TemporalField field,
244                                                         TextStyle style, Locale locale) {
245         if (chrono == IsoChronology.INSTANCE
246             || !(field instanceof ChronoField)) {
247             return getTextIterator(field, style, locale);
248         }
249
250         int fieldIndex;
251         switch ((ChronoField)field) {
252             case ERA:
253                 fieldIndex = Calendar.ERA;
254                 break;
255             case MONTH_OF_YEAR:
256                 fieldIndex = Calendar.MONTH;
257                 break;
258             case DAY_OF_WEEK:
259                 fieldIndex = Calendar.DAY_OF_WEEK;
260                 break;
261             case AMPM_OF_DAY:
262                 fieldIndex = Calendar.AM_PM;
263                 break;
264             default:
265                 return null;
266         }
267
268         int calendarStyle = (style == null) ? Calendar.ALL_STYLES : style.toCalendarStyle();
269         Map<String, Integer> map = CalendarDataUtility.retrieveJavaTimeFieldValueNames(
270             chrono.getCalendarType(), fieldIndex, calendarStyle, locale);
271         if (map == null) {
272             return null;
273         }
274         List<Entry<String, Long>> list = new ArrayList<>(map.size());
275         switch (fieldIndex) {
276             case Calendar.ERA:
277                 for (Map.Entry<String, Integer> entry : map.entrySet()) {
278                     int era = entry.getValue();
279                     if (chrono == JapaneseChronology.INSTANCE) {
280                         if (era == 0) {
281                             era = -999;

```

```

282         } else {
283             era -= 2;
284         }
285     }
286     list.add(createEntry(entry.getKey(), (long)era));
287 }
288 break;
289 case Calendar.MONTH:
290     for (Map.Entry<String, Integer> entry : map.entrySet()) {
291         list.add(createEntry(entry.getKey(), (long)(entry.getValue() + 1)));
292     }
293     break;
294 case Calendar.DAY_OF_WEEK:
295     for (Map.Entry<String, Integer> entry : map.entrySet()) {
296         list.add(createEntry(entry.getKey(), (long)toWeekDay(entry.getValue())));
297     }
298     break;
299 default:
300     for (Map.Entry<String, Integer> entry : map.entrySet()) {
301         list.add(createEntry(entry.getKey(), (long)entry.getValue()));
302     }
303     break;
304 }
305 return list.iterator();
306 }
307
308 private Object findStore(TemporalField field, Locale locale) {
309     Entry<TemporalField, Locale> key = createEntry(field, locale);
310     Object store = CACHE.get(key);
311     if (store == null) {
312         store = createStore(field, locale);
313         CACHE.putIfAbsent(key, store);
314         store = CACHE.get(key);
315     }
316     return store;
317 }
318
319 private static int toWeekDay(int calWeekDay) {
320     if (calWeekDay == Calendar.SUNDAY) {
321         return 7;
322     } else {
323         return calWeekDay - 1;
324     }
325 }
326
327 private Object createStore(TemporalField field, Locale locale) {
328     Map<TextStyle, Map<Long, String>> styleMap = new HashMap<>();
329     if (field == ERA) {
330         for (TextStyle textStyle : TextStyle.values()) {
331             if (textStyle.isStandalone()) {
332                 // Stand-alone isn't applicable to era names.
333                 continue;
334             }

```

```

335         Map<String, Integer> displayNames = CalendarDataUtility.
            retrieveJavaTimeFieldValueNames(
336             "gregory", Calendar.ERA, textStyle.toCalendarStyle(), locale);
337         if (displayNames != null) {
338             Map<Long, String> map = new HashMap<>();
339             for (Entry<String, Integer> entry : displayNames.entrySet()) {
340                 map.put((long) entry.getValue(), entry.getKey());
341             }
342             if (!map.isEmpty()) {
343                 styleMap.put(textStyle, map);
344             }
345         }
346     }
347     return new LocaleStore(styleMap);
348 }
349
350 if (field == MONTH_OF_YEAR) {
351     for (TextStyle textStyle : TextStyle.values()) {
352         Map<String, Integer> displayNames = CalendarDataUtility.
            retrieveJavaTimeFieldValueNames(
353             "gregory", Calendar.MONTH, textStyle.toCalendarStyle(), locale);
354         Map<Long, String> map = new HashMap<>();
355         if (displayNames != null) {
356             for (Entry<String, Integer> entry : displayNames.entrySet()) {
357                 map.put((long) (entry.getValue() + 1), entry.getKey());
358             }
359
360         } else {
361             // Narrow names may have duplicated names, such as "J" for January, Jun, July.
362             // Get names one by one in that case.
363             for (int month = Calendar.JANUARY; month <= Calendar.DECEMBER; month++) {
364                 String name;
365                 name = CalendarDataUtility.retrieveJavaTimeFieldName(
366                     "gregory", Calendar.MONTH, month, textStyle.toCalendarStyle(),
                        locale);
367                 if (name == null) {
368                     break;
369                 }
370                 map.put((long) (month + 1), name);
371             }
372         }
373         if (!map.isEmpty()) {
374             styleMap.put(textStyle, map);
375         }
376     }
377     return new LocaleStore(styleMap);
378 }
379
380 if (field == DAY_OF_WEEK) {
381     for (TextStyle textStyle : TextStyle.values()) {
382         Map<String, Integer> displayNames = CalendarDataUtility.
            retrieveJavaTimeFieldValueNames(
383             "gregory", Calendar.DAY_OF_WEEK, textStyle.toCalendarStyle(), locale);

```



```

384         Map<Long, String> map = new HashMap<>();
385         if (displayNames != null) {
386             for (Entry<String, Integer> entry : displayNames.entrySet()) {
387                 map.put((long)toWeekDay(entry.getValue()), entry.getKey());
388             }
389
390         } else {
391             // Narrow names may have duplicated names, such as "S" for Sunday and Saturday.
392             // Get names one by one in that case.
393             for (int wday = Calendar.SUNDAY; wday <= Calendar.SATURDAY; wday++) {
394                 String name;
395                 name = CalendarDataUtility.retrieveJavaTimeFieldValueName(
396                     "gregory", Calendar.DAY_OF_WEEK, wday, textStyle.toCalendarStyle(),
397                     locale);
398                 if (name == null) {
399                     break;
400                 }
401                 map.put((long)toWeekDay(wday), name);
402             }
403             if (!map.isEmpty()) {
404                 styleMap.put(textStyle, map);
405             }
406         }
407         return new LocaleStore(styleMap);
408     }
409
410     if (field == AMPM_OF_DAY) {
411         for (TextStyle textStyle : TextStyle.values()) {
412             if (textStyle.isStandalone()) {
413                 // Stand-alone isn't applicable to AM/PM.
414                 continue;
415             }
416             Map<String, Integer> displayNames = CalendarDataUtility.
417                 retrieveJavaTimeFieldValueNames(
418                     "gregory", Calendar.AM_PM, textStyle.toCalendarStyle(), locale);
419             if (displayNames != null) {
420                 Map<Long, String> map = new HashMap<>();
421                 for (Entry<String, Integer> entry : displayNames.entrySet()) {
422                     map.put((long)entry.getValue(), entry.getKey());
423                 }
424                 if (!map.isEmpty()) {
425                     styleMap.put(textStyle, map);
426                 }
427             }
428             return new LocaleStore(styleMap);
429         }
430
431     if (field == IsoFields.QUARTER_OF_YEAR) {
432         // The order of keys must correspond to the TextStyle.values() order.
433         final String[] keys = {
434             "QuarterNames",

```

```

435         "standalone.QuarterNames",
436         "QuarterAbbreviations",
437         "standalone.QuarterAbbreviations",
438         "QuarterNarrows",
439         "standalone.QuarterNarrows",
440     };
441     for (int i = 0; i < keys.length; i++) {
442         String[] names = getLocalizedResource(keys[i], locale);
443         if (names != null) {
444             Map<Long, String> map = new HashMap<>();
445             for (int q = 0; q < names.length; q++) {
446                 map.put((long) (q + 1), names[q]);
447             }
448             styleMap.put(TextStyle.values()[i], map);
449         }
450     }
451     return new LocaleStore(styleMap);
452 }
453
454     return ""; // null marker for map
455 }
456
457 /**
458  * Helper method to create an immutable entry.
459  *
460  * @param text the text, not null
461  * @param field the field, not null
462  * @return the entry, not null
463  */
464 private static <A, B> Entry<A, B> createEntry(A text, B field) {
465     return new SimpleImmutableEntry<>(text, field);
466 }
467
468 /**
469  * Returns the localized resource of the given key and locale, or null
470  * if no localized resource is available.
471  *
472  * @param key the key of the localized resource, not null
473  * @param locale the locale, not null
474  * @return the localized resource, or null if not available
475  * @throws NullPointerException if key or locale is null
476  */
477 @SuppressWarnings("unchecked")
478 static <T> T getLocalizedResource(String key, Locale locale) {
479     LocaleResources lr = LocaleProviderAdapter.getResourceBundleBased()
480         .getLocaleResources(locale);
481     ResourceBundle rb = lr.getJavaTimeFormatData();
482     return rb.containsKey(key) ? (T) rb.getObject(key) : null;
483 }
484
485 /**
486  * Stores the text for a single locale.
487  * <p>

```

```

488     * Some fields have a textual representation, such as day-of-week or month-of-year.
489     * These textual representations can be captured in this class for printing
490     * and parsing.
491     * <p>
492     * This class is immutable and thread-safe.
493     */
494     static final class LocaleStore {
495         /**
496          * Map of value to text.
497          */
498         private final Map<TextStyle, Map<Long, String>> valueTextMap;
499         /**
500          * Parsable data.
501          */
502         private final Map<TextStyle, List<Entry<String, Long>>> parsable;
503
504         /**
505          * Constructor.
506          *
507          * @param valueTextMap the map of values to text to store, assigned and not altered, not
508          *                      null
509          */
510         LocaleStore(Map<TextStyle, Map<Long, String>> valueTextMap) {
511             this.valueTextMap = valueTextMap;
512             Map<TextStyle, List<Entry<String, Long>>> map = new HashMap<>();
513             List<Entry<String, Long>> allList = new ArrayList<>();
514             for (Map.Entry<TextStyle, Map<Long, String>> vtmEntry : valueTextMap.entrySet()) {
515                 Map<String, Entry<String, Long>> reverse = new HashMap<>();
516                 for (Map.Entry<Long, String> entry : vtmEntry.getValue().entrySet()) {
517                     if (reverse.put(entry.getValue(), createEntry(entry.getValue(), entry.getKey()))
518                         != null) {
519                         // TODO: BUG: this has no effect
520                         continue; // not parsable, try next style
521                     }
522                 }
523                 List<Entry<String, Long>> list = new ArrayList<>(reverse.values());
524                 Collections.sort(list, COMPARATOR);
525                 map.put(vtmEntry.getKey(), list);
526                 allList.addAll(list);
527                 map.put(null, allList);
528             }
529             Collections.sort(allList, COMPARATOR);
530             this.parsable = map;
531         }
532
533         /**
534          * Gets the text for the specified field value, locale and style
535          * for the purpose of printing.
536          *
537          * @param value the value to get text for, not null
538          * @param style the style to get text for, not null
539          * @return the text for the field value, null if no text found
540          */

```

```

539     String getText(long value, TextStyle style) {
540         Map<Long, String> map = valueTextMap.get(style);
541         return map != null ? map.get(value) : null;
542     }
543
544     /**
545      * Gets an iterator of text to field for the specified style for the purpose of parsing.
546      * <p>
547      * The iterator must be returned in order from the longest text to the shortest.
548      *
549      * @param style the style to get text for, null for all parsable text
550      * @return the iterator of text to field pairs, in order from longest text to shortest text
551      *
552      * null if the style is not parsable
553      */
554     Iterator<Entry<String, Long>> getTextIterator(TextStyle style) {
555         List<Entry<String, Long>> list = parsable.get(style);
556         return list != null ? list.iterator() : null;
557     }
558 }

```

6.7 DecimalStyle.java

Secuencia 104: java/time/format/DecimalStyle.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *

```

```
29 *
30 *
31 *
32 * Copyright (c) 2008-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
62 package java.time.format;
63
64 import java.text.DecimalFormatSymbols;
65 import java.util.Collections;
66 import java.util.HashSet;
67 import java.util.Locale;
68 import java.util.Objects;
69 import java.util.Set;
70 import java.util.concurrent.ConcurrentHashMap;
71 import java.util.concurrent.ConcurrentMap;
72
73 /**
74  * Localized decimal style used in date and time formatting.
75  * <p>
76  * A significant part of dealing with dates and times is the localization.
77  * This class acts as a central point for accessing the information.
78  *
79  * @implSpec
80  * This class is immutable and thread-safe.
81  *
```

```

82  * @since 1.8
83  */
84  public final class DecimalStyle {
85
86      /**
87       * The standard set of non-localized decimal style symbols.
88       * <p>
89       * This uses standard ASCII characters for zero, positive, negative and a dot for the decimal
90       * point.
91       */
92      public static final DecimalStyle STANDARD = new DecimalStyle('0', '+', '-', '.');
93      /**
94       * The cache of DecimalStyle instances.
95       */
96      private static final ConcurrentMap<Locale, DecimalStyle> CACHE = new ConcurrentHashMap<>(16,
97          0.75f, 2);
98
99      /**
100     * The zero digit.
101     */
102     private final char zeroDigit;
103     /**
104     * The positive sign.
105     */
106     private final char positiveSign;
107     /**
108     * The negative sign.
109     */
110     private final char negativeSign;
111     /**
112     * The decimal separator.
113     */
114     private final char decimalSeparator;
115
116     //-----
117     /**
118     * Lists all the locales that are supported.
119     * <p>
120     * The locale 'en_US' will always be present.
121     *
122     * @return a Set of Locales for which localization is supported
123     */
124     public static Set<Locale> getAvailableLocales() {
125         Locale[] l = DecimalFormatSymbols.getAvailableLocales();
126         Set<Locale> locales = new HashSet<>(l.length);
127         Collections.addAll(locales, l);
128         return locales;
129     }
130
131     /**
132     * Obtains the DecimalStyle for the default
133     * {@link java.util.Locale.Category#FORMAT FORMAT} locale.
134     * <p>

```

```

133     * This method provides access to locale sensitive decimal style symbols.
134     * <p>
135     * This is equivalent to calling
136     * {@link #of(Locale)}
137     *   of(Locale.getDefault(Locale.Category.FORMAT)).
138     *
139     * @see java.util.Locale.Category#FORMAT
140     * @return the decimal style, not null
141     */
142     public static DecimalStyle ofDefaultLocale() {
143         return of(Locale.getDefault(Locale.Category.FORMAT));
144     }
145
146     /**
147     * Obtains the DecimalStyle for the specified locale.
148     * <p>
149     * This method provides access to locale sensitive decimal style symbols.
150     *
151     * @param locale the locale, not null
152     * @return the decimal style, not null
153     */
154     public static DecimalStyle of(Locale locale) {
155         Objects.requireNonNull(locale, "locale");
156         DecimalStyle info = CACHE.get(locale);
157         if (info == null) {
158             info = create(locale);
159             CACHE.putIfAbsent(locale, info);
160             info = CACHE.get(locale);
161         }
162         return info;
163     }
164
165     private static DecimalStyle create(Locale locale) {
166         DecimalFormatSymbols oldSymbols = DecimalFormatSymbols.getInstance(locale);
167         char zeroDigit = oldSymbols.getZeroDigit();
168         char positiveSign = '+';
169         char negativeSign = oldSymbols.getMinusSign();
170         char decimalSeparator = oldSymbols.getDecimalSeparator();
171         if (zeroDigit == '0' && negativeSign == '-' && decimalSeparator == '.') {
172             return STANDARD;
173         }
174         return new DecimalStyle(zeroDigit, positiveSign, negativeSign, decimalSeparator);
175     }
176
177     //-----
178     /**
179     * Restricted constructor.
180     *
181     * @param zeroChar the character to use for the digit of zero
182     * @param positiveSignChar the character to use for the positive sign
183     * @param negativeSignChar the character to use for the negative sign
184     * @param decimalPointChar the character to use for the decimal point
185     */

```

```

186     private DecimalStyle(char zeroChar, char positiveSignChar, char negativeSignChar, char
        decimalPointChar) {
187         this.zeroDigit = zeroChar;
188         this.positiveSign = positiveSignChar;
189         this.negativeSign = negativeSignChar;
190         this.decimalSeparator = decimalPointChar;
191     }
192
193     //-----
194     /**
195      * Gets the character that represents zero.
196      * <p>
197      * The character used to represent digits may vary by culture.
198      * This method specifies the zero character to use, which implies the characters for one to
        nine.
199      *
200      * @return the character for zero
201      */
202     public char getZeroDigit() {
203         return zeroDigit;
204     }
205
206     /**
207      * Returns a copy of the info with a new character that represents zero.
208      * <p>
209      * The character used to represent digits may vary by culture.
210      * This method specifies the zero character to use, which implies the characters for one to
        nine.
211      *
212      * @param zeroDigit the character for zero
213      * @return a copy with a new character that represents zero, not null
214      *
215      */
216     public DecimalStyle withZeroDigit(char zeroDigit) {
217         if (zeroDigit == this.zeroDigit) {
218             return this;
219         }
220         return new DecimalStyle(zeroDigit, positiveSign, negativeSign, decimalSeparator);
221     }
222
223     //-----
224     /**
225      * Gets the character that represents the positive sign.
226      * <p>
227      * The character used to represent a positive number may vary by culture.
228      * This method specifies the character to use.
229      *
230      * @return the character for the positive sign
231      */
232     public char getPositiveSign() {
233         return positiveSign;
234     }
235

```



```
236 /**
237  * Returns a copy of the info with a new character that represents the positive sign.
238  * <p>
239  * The character used to represent a positive number may vary by culture.
240  * This method specifies the character to use.
241  *
242  * @param positiveSign the character for the positive sign
243  * @return a copy with a new character that represents the positive sign, not null
244  */
245 public DecimalStyle withPositiveSign(char positiveSign) {
246     if (positiveSign == this.positiveSign) {
247         return this;
248     }
249     return new DecimalStyle(zeroDigit, positiveSign, negativeSign, decimalSeparator);
250 }
251
252 //-----
253 /**
254  * Gets the character that represents the negative sign.
255  * <p>
256  * The character used to represent a negative number may vary by culture.
257  * This method specifies the character to use.
258  *
259  * @return the character for the negative sign
260  */
261 public char getNegativeSign() {
262     return negativeSign;
263 }
264
265 /**
266  * Returns a copy of the info with a new character that represents the negative sign.
267  * <p>
268  * The character used to represent a negative number may vary by culture.
269  * This method specifies the character to use.
270  *
271  * @param negativeSign the character for the negative sign
272  * @return a copy with a new character that represents the negative sign, not null
273  */
274 public DecimalStyle withNegativeSign(char negativeSign) {
275     if (negativeSign == this.negativeSign) {
276         return this;
277     }
278     return new DecimalStyle(zeroDigit, positiveSign, negativeSign, decimalSeparator);
279 }
280
281 //-----
282 /**
283  * Gets the character that represents the decimal point.
284  * <p>
285  * The character used to represent a decimal point may vary by culture.
286  * This method specifies the character to use.
287  *
288  * @return the character for the decimal point
```

```

289     */
290     public char getDecimalSeparator() {
291         return decimalSeparator;
292     }
293
294     /**
295      * Returns a copy of the info with a new character that represents the decimal point.
296      * <p>
297      * The character used to represent a decimal point may vary by culture.
298      * This method specifies the character to use.
299      *
300      * @param decimalSeparator the character for the decimal point
301      * @return a copy with a new character that represents the decimal point, not null
302      */
303     public DecimalStyle withDecimalSeparator(char decimalSeparator) {
304         if (decimalSeparator == this.decimalSeparator) {
305             return this;
306         }
307         return new DecimalStyle(zeroDigit, positiveSign, negativeSign, decimalSeparator);
308     }
309
310     //-----
311     /**
312      * Checks whether the character is a digit, based on the currently set zero character.
313      *
314      * @param ch the character to check
315      * @return the value, 0 to 9, of the character, or -1 if not a digit
316      */
317     int convertToDigit(char ch) {
318         int val = ch - zeroDigit;
319         return (val >= 0 && val <= 9) ? val : -1;
320     }
321
322     /**
323      * Converts the input numeric text to the internationalized form using the zero character.
324      *
325      * @param numericText the text, consisting of digits 0 to 9, to convert, not null
326      * @return the internationalized text, not null
327      */
328     String convertNumberToI18N(String numericText) {
329         if (zeroDigit == '0') {
330             return numericText;
331         }
332         int diff = zeroDigit - '0';
333         char[] array = numericText.toCharArray();
334         for (int i = 0; i < array.length; i++) {
335             array[i] = (char) (array[i] + diff);
336         }
337         return new String(array);
338     }
339
340     //-----
341     /**

```

```

342     * Checks if this DecimalStyle is equal to another DecimalStyle.
343     *
344     * @param obj the object to check, null returns false
345     * @return true if this is equal to the other date
346     */
347     @Override
348     public boolean equals(Object obj) {
349         if (this == obj) {
350             return true;
351         }
352         if (obj instanceof DecimalStyle) {
353             DecimalStyle other = (DecimalStyle) obj;
354             return (zeroDigit == other.zeroDigit && positiveSign == other.positiveSign &&
355                 negativeSign == other.negativeSign && decimalSeparator == other.
356                     decimalSeparator);
357         }
358         return false;
359     }
360     /**
361     * A hash code for this DecimalStyle.
362     *
363     * @return a suitable hash code
364     */
365     @Override
366     public int hashCode() {
367         return zeroDigit + positiveSign + negativeSign + decimalSeparator;
368     }
369
370     //-----
371     /**
372     * Returns a string describing this DecimalStyle.
373     *
374     * @return a string description, not null
375     */
376     @Override
377     public String toString() {
378         return "DecimalStyle[" + zeroDigit + positiveSign + negativeSign + decimalSeparator + "]";
379     }
380
381 }

```

6.8 FormatStyle.java

Secuencia 105: java/time/format/FormatStyle.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *

```

```
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2008-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
```

```

62 package java.time.format;
63
64 /**
65  * Enumeration of the style of a localized date, time or date-time formatter.
66  * <p>
67  * These styles are used when obtaining a date-time style from configuration.
68  * See {@link DateTimeFormatter} and {@link DateTimeFormatterBuilder} for usage.
69  *
70  * @implSpec
71  * This is an immutable and thread-safe enum.
72  *
73  * @since 1.8
74  */
75 public enum FormatStyle {
76     // ordered from large to small
77
78     /**
79      * Full text style, with the most detail.
80      * For example, the format might be 'Tuesday, April 12, 1952 AD' or '3:30:42pm PST'.
81      */
82     FULL,
83     /**
84      * Long text style, with lots of detail.
85      * For example, the format might be 'January 12, 1952'.
86      */
87     LONG,
88     /**
89      * Medium text style, with some detail.
90      * For example, the format might be 'Jan 12, 1952'.
91      */
92     MEDIUM,
93     /**
94      * Short text style, typically numeric.
95      * For example, the format might be '12.13.52' or '3:30pm'.
96      */
97     SHORT;
98
99 }

```

6.9 Parsed.java

Secuencia 106: java/time/format/Parsed.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *

```

```
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2008-2013, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.format;
63
64  import static java.time.temporal.ChronoField.AMPM_OF_DAY;
```

```
65 import static java.time.temporal.ChronoField.CLOCK_HOUR_OF_AMPM;
66 import static java.time.temporal.ChronoField.CLOCK_HOUR_OF_DAY;
67 import static java.time.temporal.ChronoField.HOUR_OF_AMPM;
68 import static java.time.temporal.ChronoField.HOUR_OF_DAY;
69 import static java.time.temporal.ChronoField.INSTANT_SECONDS;
70 import static java.time.temporal.ChronoField.MICRO_OF_DAY;
71 import static java.time.temporal.ChronoField.MICRO_OF_SECOND;
72 import static java.time.temporal.ChronoField.MILLI_OF_DAY;
73 import static java.time.temporal.ChronoField.MILLI_OF_SECOND;
74 import static java.time.temporal.ChronoField.MINUTE_OF_DAY;
75 import static java.time.temporal.ChronoField.MINUTE_OF_HOUR;
76 import static java.time.temporal.ChronoField.NANO_OF_DAY;
77 import static java.time.temporal.ChronoField.NANO_OF_SECOND;
78 import static java.time.temporal.ChronoField.OFFSET_SECONDS;
79 import static java.time.temporal.ChronoField.SECOND_OF_DAY;
80 import static java.time.temporal.ChronoField.SECOND_OF_MINUTE;
81
82 import java.time.DateTimeException;
83 import java.time.Instant;
84 import java.time.LocalDate;
85 import java.time.LocalDateTime;
86 import java.time.Period;
87 import java.time.ZoneId;
88 import java.time.ZoneOffset;
89 import java.time.chrono.ChronoLocalDate;
90 import java.time.chrono.ChronoLocalDateTime;
91 import java.time.chrono.ChronoZonedDateTime;
92 import java.time.chrono.Chronology;
93 import java.time.temporal.ChronoField;
94 import java.time.temporal.TemporalAccessor;
95 import java.time.temporal.TemporalField;
96 import java.time.temporal.TemporalQueries;
97 import java.time.temporal.TemporalQuery;
98 import java.time.temporal.UnsupportedTemporalTypeException;
99 import java.util.HashMap;
100 import java.util.Iterator;
101 import java.util.Map;
102 import java.util.Map.Entry;
103 import java.util.Objects;
104 import java.util.Set;
105
106 /**
107  * A store of parsed data.
108  * <p>
109  * This class is used during parsing to collect the data. Part of the parsing process
110  * involves handling optional blocks and multiple copies of the data get created to
111  * support the necessary backtracking.
112  * <p>
113  * Once parsing is completed, this class can be used as the resultant {@code TemporalAccessor}.
114  * In most cases, it is only exposed once the fields have been resolved.
115  *
116  * @implSpec
117  * This class is a mutable context intended for use from a single thread.
```

```
118 * Usage of the class is thread-safe within standard parsing as a new instance of this class
119 * is automatically created for each parse and parsing is single-threaded
120 *
121 * @since 1.8
122 */
123 final class Parsed implements TemporalAccessor {
124     // some fields are accessed using package scope from DateTimeParseContext
125
126     /**
127      * The parsed fields.
128      */
129     final Map<TemporalField, Long> fieldValues = new HashMap<>();
130     /**
131      * The parsed zone.
132      */
133     ZoneId zone;
134     /**
135      * The parsed chronology.
136      */
137     Chronology chrono;
138     /**
139      * Whether a leap-second is parsed.
140      */
141     boolean leapSecond;
142     /**
143      * The resolver style to use.
144      */
145     private ResolverStyle resolverStyle;
146     /**
147      * The resolved date.
148      */
149     private ChronoLocalDate date;
150     /**
151      * The resolved time.
152      */
153     private LocalTime time;
154     /**
155      * The excess period from time-only parsing.
156      */
157     Period excessDays = Period.ZERO;
158
159     /**
160      * Creates an instance.
161      */
162     Parsed() {
163     }
164
165     /**
166      * Creates a copy.
167      */
168     Parsed copy() {
169         // only copy fields used in parsing stage
170         Parsed cloned = new Parsed();
```



```

171         cloned.fieldValues.putAll(this.fieldValues);
172         cloned.zone = this.zone;
173         cloned.chrono = this.chrono;
174         cloned.leapSecond = this.leapSecond;
175         return cloned;
176     }
177
178     //-----
179     @Override
180     public boolean isSupported(TemporalField field) {
181         if (fieldValues.containsKey(field) ||
182             (date != null && date.isSupported(field)) ||
183             (time != null && time.isSupported(field))) {
184             return true;
185         }
186         return field != null && (field instanceof ChronoField == false) && field.isSupportedBy(this);
187     }
188
189     @Override
190     public long getLong(TemporalField field) {
191         Objects.requireNonNull(field, "field");
192         Long value = fieldValues.get(field);
193         if (value != null) {
194             return value;
195         }
196         if (date != null && date.isSupported(field)) {
197             return date.getLong(field);
198         }
199         if (time != null && time.isSupported(field)) {
200             return time.getLong(field);
201         }
202         if (field instanceof ChronoField) {
203             throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
204         }
205         return field.getFrom(this);
206     }
207
208     @SuppressWarnings("unchecked")
209     @Override
210     public <R> R query(TemporalQuery<R> query) {
211         if (query == TemporalQueries.zoneId()) {
212             return (R) zone;
213         } else if (query == TemporalQueries.chronology()) {
214             return (R) chrono;
215         } else if (query == TemporalQueries.localDate()) {
216             return (R) (date != null ? LocalDate.from(date) : null);
217         } else if (query == TemporalQueries.localTime()) {
218             return (R) time;
219         } else if (query == TemporalQueries.zone() || query == TemporalQueries.offset()) {
220             return query.queryFrom(this);
221         } else if (query == TemporalQueries.precision()) {
222             return null; // not a complete date/time

```

```

223     }
224     // inline TemporalAccessor.super.query(query) as an optimization
225     // non-JDK classes are not permitted to make this optimization
226     return query.queryFrom(this);
227 }
228
229 //-----
230 /**
231  * Resolves the fields in this context.
232  *
233  * @param resolverStyle the resolver style, not null
234  * @param resolverFields the fields to use for resolving, null for all fields
235  * @return this, for method chaining
236  * @throws DateTimeException if resolving one field results in a value for
237  *         another field that is in conflict
238  */
239 TemporalAccessor resolve(ResolverStyle resolverStyle, Set<TemporalField> resolverFields) {
240     if (resolverFields != null) {
241         fieldValues.keySet().retainAll(resolverFields);
242     }
243     this.resolverStyle = resolverStyle;
244     resolveFields();
245     resolveTimeLenient();
246     crossCheck();
247     resolvePeriod();
248     resolveFractional();
249     resolveInstant();
250     return this;
251 }
252
253 //-----
254 private void resolveFields() {
255     // resolve ChronoField
256     resolveInstantFields();
257     resolveDateFields();
258     resolveTimeFields();
259
260     // if any other fields, handle them
261     // any lenient date resolution should return epoch-day
262     if (fieldValues.size() > 0) {
263         int changedCount = 0;
264         outer:
265         while (changedCount < 50) {
266             for (Map.Entry<TemporalField, Long> entry : fieldValues.entrySet()) {
267                 TemporalField targetField = entry.getKey();
268                 TemporalAccessor resolvedObject = targetField.resolve(fieldValues, this,
269                     resolverStyle);
270                 if (resolvedObject != null) {
271                     if (resolvedObject instanceof ChronoZonedDateTime) {
272                         ChronoZonedDateTime<?> czdt = (ChronoZonedDateTime<?>) resolvedObject;
273                         if (zone == null) {
274                             zone = czdt.getZone();
275                         } else if (zone.equals(czdt.getZone()) == false) {

```

```

275         throw new DateTimeException("ChronoZonedDateTime must use the
276             effective parsed zone: " + zone);
277     }
278     resolvedObject = czdt.toLocalDateTime();
279 }
280 if (resolvedObject instanceof ChronoLocalDateTime) {
281     ChronoLocalDateTime<?> cldt = (ChronoLocalDateTime<?>) resolvedObject;
282     updateCheckConflict(cldt.toLocalTime(), Period.ZERO);
283     updateCheckConflict(cldt.toLocalDate());
284     changedCount++;
285     continue outer; // have to restart to avoid concurrent modification
286 }
287 if (resolvedObject instanceof ChronoLocalDate) {
288     updateCheckConflict((ChronoLocalDate) resolvedObject);
289     changedCount++;
290     continue outer; // have to restart to avoid concurrent modification
291 }
292 if (resolvedObject instanceof LocalTime) {
293     updateCheckConflict((LocalTime) resolvedObject, Period.ZERO);
294     changedCount++;
295     continue outer; // have to restart to avoid concurrent modification
296 }
297 throw new DateTimeException("Method resolve() can only return
298     ChronoZonedDateTime, " +
299     "ChronoLocalDateTime, ChronoLocalDate or LocalTime");
300 } else if (fieldValues.containsKey(targetField) == false) {
301     changedCount++;
302     continue outer; // have to restart to avoid concurrent modification
303 }
304 }
305 break;
306 }
307 if (changedCount == 50) { // catch infinite loops
308     throw new DateTimeException("One of the parsed fields has an incorrectly
309         implemented resolve method");
310 }
311 // if something changed then have to redo ChronoField resolve
312 if (changedCount > 0) {
313     resolveInstantFields();
314     resolveDateFields();
315     resolveTimeFields();
316 }
317 }
318 }
319
320 private void updateCheckConflict(TemporalField targetField, TemporalField changeField, Long
321     changeValue) {
322     Long old = fieldValues.put(changeField, changeValue);
323     if (old != null && old.longValue() != changeValue.longValue()) {
324         throw new DateTimeException("Conflict found: " + changeField + " " + old +
325             " differs from " + changeField + " " + changeValue +
326             " while resolving " + targetField);
327     }
328 }

```

```

324     }
325
326     //-----
327     private void resolveInstantFields() {
328         // resolve parsed instant seconds to date and time if zone available
329         if (fieldValues.containsKey(INSTANT_SECONDS)) {
330             if (zone != null) {
331                 resolveInstantFields0(zone);
332             } else {
333                 Long offsetSecs = fieldValues.get(OFFSET_SECONDS);
334                 if (offsetSecs != null) {
335                     ZoneOffset offset = ZoneOffset.ofTotalSeconds(offsetSecs.intValue());
336                     resolveInstantFields0(offset);
337                 }
338             }
339         }
340     }
341
342     private void resolveInstantFields0(ZoneId selectedZone) {
343         Instant instant = Instant.ofEpochSecond(fieldValues.remove(INSTANT_SECONDS));
344         ChronoZonedDateTime<?> zdt = chrono.zonedDateTime(instant, selectedZone);
345         updateCheckConflict(zdt.toLocalDate());
346         updateCheckConflict(INSTANT_SECONDS, SECOND_OF_DAY, (long) zdt.toLocalTime().toSecondOfDay());
347     }
348
349     //-----
350     private void resolveDateFields() {
351         updateCheckConflict(chrono.resolveDate(fieldValues, resolverStyle));
352     }
353
354     private void updateCheckConflict(ChronoLocalDate cld) {
355         if (date != null) {
356             if (cld != null && date.equals(cld) == false) {
357                 throw new DateTimeException("Conflict found: Fields resolved to two different dates
358                     : " + date + " " + cld);
359             }
360         } else if (cld != null) {
361             if (chrono.equals(cld.getChronology()) == false) {
362                 throw new DateTimeException("ChronoLocalDate must use the effective parsed
363                     chronology: " + chrono);
364             }
365             date = cld;
366         }
367     }
368
369     //-----
370     private void resolveTimeFields() {
371         // simplify fields
372         if (fieldValues.containsKey(CLOCK_HOUR_OF_DAY)) {
373             // lenient allows anything, smart allows 0-24, strict allows 1-24
374             long ch = fieldValues.remove(CLOCK_HOUR_OF_DAY);
375             if (resolverStyle == ResolverStyle.STRICT || (resolverStyle == ResolverStyle.SMART &&

```

```

        ch != 0)) {
374     CLOCK_HOUR_OF_DAY.checkValidValue(ch);
375 }
376 updateCheckConflict(CLOCK_HOUR_OF_DAY, HOUR_OF_DAY, ch == 24 ? 0 : ch);
377 }
378 if (fieldValues.containsKey(CLOCK_HOUR_OF_AMPM)) {
379     // lenient allows anything, smart allows 0-12, strict allows 1-12
380     long ch = fieldValues.remove(CLOCK_HOUR_OF_AMPM);
381     if (resolverStyle == ResolverStyle.STRICT || (resolverStyle == ResolverStyle.SMART &&
        ch != 0)) {
382         CLOCK_HOUR_OF_AMPM.checkValidValue(ch);
383     }
384     updateCheckConflict(CLOCK_HOUR_OF_AMPM, HOUR_OF_AMPM, ch == 12 ? 0 : ch);
385 }
386 if (fieldValues.containsKey(AMPM_OF_DAY) && fieldValues.containsKey(HOUR_OF_AMPM)) {
387     long ap = fieldValues.remove(AMPM_OF_DAY);
388     long hap = fieldValues.remove(HOUR_OF_AMPM);
389     if (resolverStyle == ResolverStyle.LENIENT) {
390         updateCheckConflict(AMPM_OF_DAY, HOUR_OF_DAY, Math.addExact(Math.multiplyExact(ap,
        12), hap));
391     } else { // STRICT or SMART
392         AMPM_OF_DAY.checkValidValue(ap);
393         HOUR_OF_AMPM.checkValidValue(ap);
394         updateCheckConflict(AMPM_OF_DAY, HOUR_OF_DAY, ap * 12 + hap);
395     }
396 }
397 if (fieldValues.containsKey(NANO_OF_DAY)) {
398     long nod = fieldValues.remove(NANO_OF_DAY);
399     if (resolverStyle != ResolverStyle.LENIENT) {
400         NANO_OF_DAY.checkValidValue(nod);
401     }
402     updateCheckConflict(NANO_OF_DAY, HOUR_OF_DAY, nod / 3600_000_000_000L);
403     updateCheckConflict(NANO_OF_DAY, MINUTE_OF_HOUR, (nod / 60_000_000_000L) % 60);
404     updateCheckConflict(NANO_OF_DAY, SECOND_OF_MINUTE, (nod / 1_000_000_000L) % 60);
405     updateCheckConflict(NANO_OF_DAY, NANO_OF_SECOND, nod % 1_000_000_000L);
406 }
407 if (fieldValues.containsKey(MICRO_OF_DAY)) {
408     long cod = fieldValues.remove(MICRO_OF_DAY);
409     if (resolverStyle != ResolverStyle.LENIENT) {
410         MICRO_OF_DAY.checkValidValue(cod);
411     }
412     updateCheckConflict(MICRO_OF_DAY, SECOND_OF_DAY, cod / 1_000_000L);
413     updateCheckConflict(MICRO_OF_DAY, MICRO_OF_SECOND, cod % 1_000_000L);
414 }
415 if (fieldValues.containsKey(MILLI_OF_DAY)) {
416     long lod = fieldValues.remove(MILLI_OF_DAY);
417     if (resolverStyle != ResolverStyle.LENIENT) {
418         MILLI_OF_DAY.checkValidValue(lod);
419     }
420     updateCheckConflict(MILLI_OF_DAY, SECOND_OF_DAY, lod / 1_000);
421     updateCheckConflict(MILLI_OF_DAY, MILLI_OF_SECOND, lod % 1_000);
422 }
423 if (fieldValues.containsKey(SECOND_OF_DAY)) {

```

```

424         long sod = fieldValues.remove(SECOND_OF_DAY);
425         if (resolverStyle != ResolverStyle.LENIENT) {
426             SECOND_OF_DAY.checkValidValue(sod);
427         }
428         updateCheckConflict(SECOND_OF_DAY, HOUR_OF_DAY, sod / 3600);
429         updateCheckConflict(SECOND_OF_DAY, MINUTE_OF_HOUR, (sod / 60) % 60);
430         updateCheckConflict(SECOND_OF_DAY, SECOND_OF_MINUTE, sod % 60);
431     }
432     if (fieldValues.containsKey(MINUTE_OF_DAY)) {
433         long mod = fieldValues.remove(MINUTE_OF_DAY);
434         if (resolverStyle != ResolverStyle.LENIENT) {
435             MINUTE_OF_DAY.checkValidValue(mod);
436         }
437         updateCheckConflict(MINUTE_OF_DAY, HOUR_OF_DAY, mod / 60);
438         updateCheckConflict(MINUTE_OF_DAY, MINUTE_OF_HOUR, mod % 60);
439     }
440
441     // combine partial second fields strictly, leaving lenient expansion to later
442     if (fieldValues.containsKey(NANO_OF_SECOND)) {
443         long nos = fieldValues.get(NANO_OF_SECOND);
444         if (resolverStyle != ResolverStyle.LENIENT) {
445             NANO_OF_SECOND.checkValidValue(nos);
446         }
447         if (fieldValues.containsKey(MICRO_OF_SECOND)) {
448             long cos = fieldValues.remove(MICRO_OF_SECOND);
449             if (resolverStyle != ResolverStyle.LENIENT) {
450                 MICRO_OF_SECOND.checkValidValue(cos);
451             }
452             nos = cos * 1000 + (nos % 1000);
453             updateCheckConflict(MICRO_OF_SECOND, NANO_OF_SECOND, nos);
454         }
455         if (fieldValues.containsKey(MILLI_OF_SECOND)) {
456             long los = fieldValues.remove(MILLI_OF_SECOND);
457             if (resolverStyle != ResolverStyle.LENIENT) {
458                 MILLI_OF_SECOND.checkValidValue(los);
459             }
460             updateCheckConflict(MILLI_OF_SECOND, NANO_OF_SECOND, los * 1_000_000L + (nos % 1_000_000L));
461         }
462     }
463
464     // convert to time if all four fields available (optimization)
465     if (fieldValues.containsKey(HOUR_OF_DAY) && fieldValues.containsKey(MINUTE_OF_HOUR) &&
466         fieldValues.containsKey(SECOND_OF_MINUTE) && fieldValues.containsKey(NANO_OF_SECOND)) {
467         long hod = fieldValues.remove(HOUR_OF_DAY);
468         long moh = fieldValues.remove(MINUTE_OF_HOUR);
469         long som = fieldValues.remove(SECOND_OF_MINUTE);
470         long nos = fieldValues.remove(NANO_OF_SECOND);
471         resolveTime(hod, moh, som, nos);
472     }
473 }
474

```

```

475 private void resolveTimeLenient() {
476     // leniently create a time from incomplete information
477     // done after everything else as it creates information from nothing
478     // which would break updateCheckConflict(field)
479
480     if (time == null) {
481         // NANO_OF_SECOND merged with MILLI/MICRO above
482         if (fieldValues.containsKey(MILLI_OF_SECOND)) {
483             long los = fieldValues.remove(MILLI_OF_SECOND);
484             if (fieldValues.containsKey(MICRO_OF_SECOND)) {
485                 // merge milli-of-second and micro-of-second for better error message
486                 long cos = los * 1_000 + (fieldValues.get(MICRO_OF_SECOND) % 1_000);
487                 updateCheckConflict(MILLI_OF_SECOND, MICRO_OF_SECOND, cos);
488                 fieldValues.remove(MICRO_OF_SECOND);
489                 fieldValues.put(NANO_OF_SECOND, cos * 1_000L);
490             } else {
491                 // convert milli-of-second to nano-of-second
492                 fieldValues.put(NANO_OF_SECOND, los * 1_000_000L);
493             }
494         } else if (fieldValues.containsKey(MICRO_OF_SECOND)) {
495             // convert micro-of-second to nano-of-second
496             long cos = fieldValues.remove(MICRO_OF_SECOND);
497             fieldValues.put(NANO_OF_SECOND, cos * 1_000L);
498         }
499
500         // merge hour/minute/second/nano leniently
501         Long hod = fieldValues.get(HOUR_OF_DAY);
502         if (hod != null) {
503             Long moh = fieldValues.get(MINUTE_OF_HOUR);
504             Long som = fieldValues.get(SECOND_OF_MINUTE);
505             Long nos = fieldValues.get(NANO_OF_SECOND);
506
507             // check for invalid combinations that cannot be defaulted
508             if ((moh == null && (som != null || nos != null)) ||
509                 (moh != null && som == null && nos != null)) {
510                 return;
511             }
512
513             // default as necessary and build time
514             long mohVal = (moh != null ? moh : 0);
515             long somVal = (som != null ? som : 0);
516             long nosVal = (nos != null ? nos : 0);
517             resolveTime(hod, mohVal, somVal, nosVal);
518             fieldValues.remove(HOUR_OF_DAY);
519             fieldValues.remove(MINUTE_OF_HOUR);
520             fieldValues.remove(SECOND_OF_MINUTE);
521             fieldValues.remove(NANO_OF_SECOND);
522         }
523     }
524
525     // validate remaining
526     if (resolverStyle != ResolverStyle.LENIENT && fieldValues.size() > 0) {
527         for (Entry<TemporalField, Long> entry : fieldValues.entrySet()) {

```

```

528         TemporalField field = entry.getKey();
529         if (field instanceof ChronoField && field.isTimeBased()) {
530             ((ChronoField) field).checkValidValue(entry.getValue());
531         }
532     }
533 }
534 }
535
536 private void resolveTime(long hod, long moh, long som, long nos) {
537     if (resolverStyle == ResolverStyle.LENIENT) {
538         long totalNanos = Math.multiplyExact(hod, 3600_000_000_000L);
539         totalNanos = Math.addExact(totalNanos, Math.multiplyExact(moh, 60_000_000_000L));
540         totalNanos = Math.addExact(totalNanos, Math.multiplyExact(som, 1_000_000_000L));
541         totalNanos = Math.addExact(totalNanos, nos);
542         int excessDays = (int) Math.floorDiv(totalNanos, 86400_000_000_000L); // safe int cast
543         long nod = Math.floorMod(totalNanos, 86400_000_000_000L);
544         updateCheckConflict(LocalTime.ofNanoOfDay(nod), Period.ofDays(excessDays));
545     } else { // STRICT or SMART
546         int mohVal = MINUTE_OF_HOUR.checkValidIntValue(moh);
547         int nosVal = NANO_OF_SECOND.checkValidIntValue(nos);
548         // handle 24:00 end of day
549         if (resolverStyle == ResolverStyle.SMART && hod == 24 && mohVal == 0 && som == 0 &&
550             nosVal == 0) {
551             updateCheckConflict(LocalTime.MIDNIGHT, Period.ofDays(1));
552         } else {
553             int hodVal = HOUR_OF_DAY.checkValidIntValue(hod);
554             int somVal = SECOND_OF_MINUTE.checkValidIntValue(som);
555             updateCheckConflict(LocalTime.of(hodVal, mohVal, somVal, nosVal), Period.ZERO);
556         }
557     }
558 }
559
560 private void resolvePeriod() {
561     // add whole days if we have both date and time
562     if (date != null && time != null && excessDays.isZero() == false) {
563         date = date.plus(excessDays);
564         excessDays = Period.ZERO;
565     }
566 }
567
568 private void resolveFractional() {
569     // ensure fractional seconds available as ChronoField requires
570     // resolveTimeLenient() will have merged MICRO_OF_SECOND/MILLI_OF_SECOND to NANO_OF_SECOND
571     if (time == null &&
572         (fieldValues.containsKey(INSTANT_SECONDS) ||
573          fieldValues.containsKey(SECOND_OF_DAY) ||
574          fieldValues.containsKey(SECOND_OF_MINUTE))) {
575         if (fieldValues.containsKey(NANO_OF_SECOND)) {
576             long nos = fieldValues.get(NANO_OF_SECOND);
577             fieldValues.put(MICRO_OF_SECOND, nos / 1000);
578             fieldValues.put(MILLI_OF_SECOND, nos / 1000000);
579         } else {
580             fieldValues.put(NANO_OF_SECOND, 0L);
581         }
582     }
583 }

```



```

580         fieldValues.put(MICRO_OF_SECOND, 0L);
581         fieldValues.put(MILLI_OF_SECOND, 0L);
582     }
583 }
584 }
585
586 private void resolveInstant() {
587     // add instant seconds if we have date, time and zone
588     if (date != null && time != null) {
589         if (zone != null) {
590             long instant = date.atTime(time).atZone(zone).getLong(ChronoField.INSTANT_SECONDS);
591             fieldValues.put(INSTANT_SECONDS, instant);
592         } else {
593             Long offsetSecs = fieldValues.get(OFFSET_SECONDS);
594             if (offsetSecs != null) {
595                 ZoneOffset offset = ZoneOffset.ofTotalSeconds(offsetSecs.intValue());
596                 long instant = date.atTime(time).atZone(offset).getLong(ChronoField.
597                     INSTANT_SECONDS);
598                 fieldValues.put(INSTANT_SECONDS, instant);
599             }
600         }
601     }
602 }
603
604 private void updateCheckConflict(LocalTime timeToSet, Period periodToSet) {
605     if (time != null) {
606         if (time.equals(timeToSet) == false) {
607             throw new DateTimeException("Conflict found: Fields resolved to different times: "
608                 + time + " " + timeToSet);
609         }
610         if (excessDays.isZero() == false && periodToSet.isZero() == false && excessDays.equals(
611             periodToSet) == false) {
612             throw new DateTimeException("Conflict found: Fields resolved to different excess
613                 periods: " + excessDays + " " + periodToSet);
614         } else {
615             excessDays = periodToSet;
616         }
617     } else {
618         time = timeToSet;
619         excessDays = periodToSet;
620     }
621 }
622
623 //-----
624 private void crossCheck() {
625     // only cross-check date, time and date-time
626     // avoid object creation if possible
627     if (date != null) {
628         crossCheck(date);
629     }
630     if (time != null) {
631         crossCheck(time);
632         if (date != null && fieldValues.size() > 0) {

```

```

629         crossCheck(date.atTime(time));
630     }
631 }
632 }
633
634 private void crossCheck(TemporalAccessor target) {
635     for (Iterator<Entry<TemporalField, Long>> it = fieldValues.entrySet().iterator(); it.
        hasNext(); ) {
636         Entry<TemporalField, Long> entry = it.next();
637         TemporalField field = entry.getKey();
638         if (target.isSupported(field)) {
639             long val1;
640             try {
641                 val1 = target.getLong(field);
642             } catch (RuntimeException ex) {
643                 continue;
644             }
645             long val2 = entry.getValue();
646             if (val1 != val2) {
647                 throw new DateTimeException("Conflict found: Field " + field + " " + val1 +
648                     " differs from " + field + " " + val2 + " derived from " + target);
649             }
650             it.remove();
651         }
652     }
653 }
654
655 //-----
656 @Override
657 public String toString() {
658     StringBuilder buf = new StringBuilder(64);
659     buf.append(fieldValues).append(',').append(chrono);
660     if (zone != null) {
661         buf.append(',').append(zone);
662     }
663     if (date != null || time != null) {
664         buf.append(" resolved to ");
665         if (date != null) {
666             buf.append(date);
667             if (time != null) {
668                 buf.append('T').append(time);
669             }
670         } else {
671             buf.append(time);
672         }
673     }
674     return buf.toString();
675 }
676
677 }

```

6.10 ResolverStyle.java

Secuencia 107: java/time/format/ResolverStyle.java

```
1  /*
2  * Copyright (c) 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2008-2013, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
```

```
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.format;
63
64  /**
65   * Enumeration of different ways to resolve dates and times.
66   * <p>
67   * Parsing a text string occurs in two phases.
68   * Phase 1 is a basic text parse according to the fields added to the builder.
69   * Phase 2 resolves the parsed field-value pairs into date and/or time objects.
70   * This style is used to control how phase 2, resolving, happens.
71   *
72   * @implSpec
73   * This is an immutable and thread-safe enum.
74   *
75   * @since 1.8
76   */
77  public enum ResolverStyle {
78
79      /**
80       * Style to resolve dates and times strictly.
81       * <p>
82       * Using strict resolution will ensure that all parsed values are within
83       * the outer range of valid values for the field. Individual fields may
84       * be further processed for strictness.
85       * <p>
86       * For example, resolving year-month and day-of-month in the ISO calendar
87       * system using strict mode will ensure that the day-of-month is valid
88       * for the year-month, rejecting invalid values.
89       */
90      STRICT,
91
92      /**
93       * Style to resolve dates and times in a smart, or intelligent, manner.
94       * <p>
95       * Using smart resolution will perform the sensible default for each
96       * field, which may be the same as strict, the same as lenient, or a third
97       * behavior. Individual fields will interpret this differently.
98       * <p>
99       * For example, resolving year-month and day-of-month in the ISO calendar
100      * system using smart mode will ensure that the day-of-month is from
101      * 1 to 31, converting any value beyond the last valid day-of-month to be
102      * the last valid day-of-month.
103      */
104      SMART,
105
106      /**
107       * Style to resolve dates and times leniently.
```

```
106     * <p>
107     * Using lenient resolution will resolve the values in an appropriate
108     * lenient manner. Individual fields will interpret this differently.
109     * <p>
110     * For example, lenient mode allows the month in the ISO calendar system
111     * to be outside the range 1 to 12.
112     * For example, month 15 is treated as being 3 months after month 12.
113     */
114     LENIENT;
115
116 }
```

6.11 SignStyle.java

Secuencia 108: java/time/format/SignStyle.java

```
1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2008-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
```

```
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
62 package java.time.format;
63
64 /**
65  * Enumeration of ways to handle the positive/negative sign.
66  * <p>
67  * The formatting engine allows the positive and negative signs of numbers
68  * to be controlled using this enum.
69  * See {@link DateTimeFormatterBuilder} for usage.
70  *
71  * @implSpec
72  * This is an immutable and thread-safe enum.
73  *
74  * @since 1.8
75  */
76 public enum SignStyle {
77
78     /**
79      * Style to output the sign only if the value is negative.
80      * <p>
81      * In strict parsing, the negative sign will be accepted and the positive sign rejected.
82      * In lenient parsing, any sign will be accepted.
83      */
84     NORMAL,
85
86     /**
87      * Style to always output the sign, where zero will output '+'.
88      * <p>
89      * In strict parsing, the absence of a sign will be rejected.
90      * In lenient parsing, any sign will be accepted, with the absence
91      * of a sign treated as a positive number.
92      */
93 }
```

```

92     ALWAYS,
93     /**
94      * Style to never output sign, only outputting the absolute value.
95      * <p>
96      * In strict parsing, any sign will be rejected.
97      * In lenient parsing, any sign will be accepted unless the width is fixed.
98      */
99     NEVER,
100    /**
101     * Style to block negative values, throwing an exception on printing.
102     * <p>
103     * In strict parsing, any sign will be rejected.
104     * In lenient parsing, any sign will be accepted unless the width is fixed.
105     */
106    NOT_NEGATIVE,
107    /**
108     * Style to always output the sign if the value exceeds the pad width.
109     * A negative value will always output the '-' sign.
110     * <p>
111     * In strict parsing, the sign will be rejected unless the pad width is exceeded.
112     * In lenient parsing, any sign will be accepted, with the absence
113     * of a sign treated as a positive number.
114     */
115    EXCEEDS_PAD;
116
117    /**
118     * Parse helper.
119     *
120     * @param positive true if positive sign parsed, false for negative sign
121     * @param strict true if strict, false if lenient
122     * @param fixedWidth true if fixed width, false if not
123     * @return
124     */
125    boolean parse(boolean positive, boolean strict, boolean fixedWidth) {
126        switch (ordinal()) {
127            case 0: // NORMAL
128                // valid if negative or (positive and lenient)
129                return !positive || !strict;
130            case 1: // ALWAYS
131            case 4: // EXCEEDS_PAD
132                return true;
133            default:
134                // valid if lenient and not fixed width
135                return !strict && !fixedWidth;
136        }
137    }
138
139 }

```

6.12 TextStyle.java

Secuencia 109: java/time/format/TextStyle.java

```
1  /*
```

```
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2008-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
```



```
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.format;
63
64  import java.util.Calendar;
65
66  /**
67   * Enumeration of the style of text formatting and parsing.
68   * <p>
69   * Text styles define three sizes for the formatted text - 'full', 'short' and 'narrow'.
70   * Each of these three sizes is available in both 'standard' and 'stand-alone' variations.
71   * <p>
72   * The difference between the three sizes is obvious in most languages.
73   * For example, in English the 'full' month is 'January', the 'short' month is 'Jan'
74   * and the 'narrow' month is 'J'. Note that the narrow size is often not unique.
75   * For example, 'January', 'June' and 'July' all have the 'narrow' text 'J'.
76   * <p>
77   * The difference between the 'standard' and 'stand-alone' forms is trickier to describe
78   * as there is no difference in English. However, in other languages there is a difference
79   * in the word used when the text is used alone, as opposed to in a complete date.
80   * For example, the word used for a month when used alone in a date picker is different
81   * to the word used for month in association with a day and year in a date.
82   *
83   * @implSpec
84   * This is immutable and thread-safe enum.
85   */
86  public enum TextStyle {
87      // ordered from large to small
88      // ordered so that bit 0 of the ordinal indicates stand-alone.
89
90      /**
91       * Full text, typically the full description.
92       * For example, day-of-week Monday might output "Monday".
93       */
94      FULL(Calendar.LONG_FORMAT, 0),
95      /**
96       * Full text for stand-alone use, typically the full description.
97       * For example, day-of-week Monday might output "Monday".
98       */
99      FULL_STANDALONE(Calendar.LONG_STANDALONE, 0),
100     /**
101      * Short text, typically an abbreviation.
102      * For example, day-of-week Monday might output "Mon".
103      */
104     SHORT(Calendar.SHORT_FORMAT, 1),
105     /**
106      * Short text for stand-alone use, typically an abbreviation.
107      * For example, day-of-week Monday might output "Mon".
108      */
109     SHORT_STANDALONE(Calendar.SHORT_STANDALONE, 1)
```

```

108     */
109     SHORT_STANDALONE(Calendar.SHORT_STANDALONE, 1),
110     /**
111      * Narrow text, typically a single letter.
112      * For example, day-of-week Monday might output "M".
113      */
114     NARROW(Calendar.NARROW_FORMAT, 1),
115     /**
116      * Narrow text for stand-alone use, typically a single letter.
117      * For example, day-of-week Monday might output "M".
118      */
119     NARROW_STANDALONE(Calendar.NARROW_STANDALONE, 1);
120
121     private final int calendarStyle;
122     private final int zoneNameStyleIndex;
123
124     private TextStyle(int calendarStyle, int zoneNameStyleIndex) {
125         this.calendarStyle = calendarStyle;
126         this.zoneNameStyleIndex = zoneNameStyleIndex;
127     }
128
129     /**
130      * Returns true if the Style is a stand-alone style.
131      * @return true if the style is a stand-alone style.
132      */
133     public boolean isStandalone() {
134         return (ordinal() & 1) == 1;
135     }
136
137     /**
138      * Returns the stand-alone style with the same size.
139      * @return the stand-alone style with the same size
140      */
141     public TextStyle asStandalone() {
142         return TextStyle.values()[ordinal() | 1];
143     }
144
145     /**
146      * Returns the normal style with the same size.
147      *
148      * @return the normal style with the same size
149      */
150     public TextStyle asNormal() {
151         return TextStyle.values()[ordinal() & ~1];
152     }
153
154     /**
155      * Returns the {@code Calendar} style corresponding to this {@code TextStyle}.
156      *
157      * @return the corresponding {@code Calendar} style
158      */
159     int toCalendarStyle() {
160         return calendarStyle;

```

```

161     }
162
163     /**
164      * Returns the relative index value to an element of the {@link
165      * java.text.DateFormatSymbols#getZoneStrings() DateFormatSymbols.getZoneStrings()}
166      * value, 0 for long names and 1 for short names (abbreviations). Note that these values
167      * do not correspond to the {@link java.util.TimeZone#LONG} and {@link
168      * java.util.TimeZone#SHORT} values.
169      *
170      * @return the relative index value to time zone names array
171      */
172     int zoneNameStyleIndex() {
173         return zoneNameStyleIndex;
174     }
175 }

```

6.13 ZoneName.java

Secuencia 110: java/time/format/ZoneName.java

```

1  /*
2  * Copyright (c) 2013, 2016, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25 package java.time.format;
26
27 import java.util.HashMap;
28 import java.util.Locale;
29 import java.util.Map;
30
31 /**
32  * A helper class to map a zone name to metazone and back to the
33  * appropriate zone id for the particular locale.
34  * <p>

```

```

35  * The zid->metazone mappings are based on CLDR metaZones.xml.
36  * The alias mappings are based on Link entries in tzdb data files.
37  */
38  class ZoneName {
39
40      public static String toZid(String zid, Locale locale) {
41          String mzone = zidToMzone.get(zid);
42          if (mzone == null && aliases.containsKey(zid)) {
43              zid = aliases.get(zid);
44              mzone = zidToMzone.get(zid);
45          }
46          if (mzone != null) {
47              Map<String, String> map = mzoneToZidL.get(mzone);
48              if (map != null && map.containsKey(locale.getCountry())) {
49                  zid = map.get(locale.getCountry());
50              } else {
51                  zid = mzoneToZid.get(mzone);
52              }
53          }
54          return toZid(zid);
55      }
56
57      public static String toZid(String zid) {
58          if (aliases.containsKey(zid)) {
59              return aliases.get(zid);
60          }
61          return zid;
62      }
63
64      private static final String[] zidMap = new String[] {
65          "Pacific/Rarotonga", "Cook", "Pacific/Rarotonga",
66          "Europe/Tirane", "Europe_Central", "Europe/Paris",
67          "America/Recife", "Brasilia", "America/Sao_Paulo",
68          "America/Argentina/San_Juan", "Argentina", "America/Buenos_Aires",
69          "Asia/Kolkata", "India", "Asia/Calcutta",
70          "America/Guayaquil", "Ecuador", "America/Guayaquil",
71          "Europe/Samara", "Moscow", "Europe/Moscow",
72          "Indian/Antananarivo", "Africa_Eastern", "Africa/Nairobi",
73          "America/Santa_Isabel", "America_Pacific", "America/Los_Angeles",
74          "America/Montserrat", "Atlantic", "America/Halifax",
75          "Pacific/Port_Moresby", "Papua_New_Guinea", "Pacific/Port_Moresby",
76          "Europe/Paris", "Europe_Central", "Europe/Paris",
77          "America/Argentina/Salta", "Argentina", "America/Buenos_Aires",
78          "Asia/Omsk", "Omsk", "Asia/Omsk",
79          "Africa/Ceuta", "Europe_Central", "Europe/Paris",
80          "America/Argentina/San_Luis", "Argentina_Western", "America/Argentina/San_Luis",
81          "America/Atikokan", "America_Eastern", "America/New_York",
82          "Asia/Vladivostok", "Vladivostok", "Asia/Vladivostok",
83          "America/Argentina/Jujuy", "Argentina", "America/Buenos_Aires",
84          "Asia/Almaty", "Kazakhstan_Eastern", "Asia/Almaty",
85          "Atlantic/Canary", "Europe_Western", "Atlantic/Canary",
86          "Asia/Bangkok", "Indochina", "Asia/Saigon",
87          "America/Caracas", "Venezuela", "America/Caracas",

```

```
88      "Australia/Hobart", "Australia_Eastern", "Australia/Sydney",
89      "America/Havana", "Cuba", "America/Havana",
90      "Africa/Malabo", "Africa_Western", "Africa/Lagos",
91      "Australia/Lord_Howe", "Lord_Howe", "Australia/Lord_Howe",
92      "Pacific/Fakaofu", "Tokelau", "Pacific/Fakaofu",
93      "America/Matamoros", "America_Central", "America/Chicago",
94      "America/Guadeloupe", "Atlantic", "America/Halifax",
95      "Europe/Helsinki", "Europe_Eastern", "Europe/Bucharest",
96      "Asia/Calcutta", "India", "Asia/Calcutta",
97      "Africa/Kinshasa", "Africa_Western", "Africa/Lagos",
98      "America/Miquelon", "Pierre_Miquelon", "America/Miquelon",
99      "Europe/Athens", "Europe_Eastern", "Europe/Bucharest",
100     "Asia/Novosibirsk", "Novosibirsk", "Asia/Novosibirsk",
101     "Indian/Cocos", "Cocos", "Indian/Cocos",
102     "Africa/Bujumbura", "Africa_Central", "Africa/Maputo",
103     "Europe/Mariehamn", "Europe_Eastern", "Europe/Bucharest",
104     "America/Winnipeg", "America_Central", "America/Chicago",
105     "America/Buenos_Aires", "Argentina", "America/Buenos_Aires",
106     "America/Yellowknife", "America_Mountain", "America/Denver",
107     "Pacific/Midway", "Samoa", "Pacific/Apia",
108     "Africa/Dar_es_Salaam", "Africa_Eastern", "Africa/Nairobi",
109     "Pacific/Tahiti", "Tahiti", "Pacific/Tahiti",
110     "Asia/Gaza", "Europe_Eastern", "Europe/Bucharest",
111     "Australia/Lindeman", "Australia_Eastern", "Australia/Sydney",
112     "Europe/Kaliningrad", "Europe_Eastern", "Europe/Bucharest",
113     "Europe/Bucharest", "Europe_Eastern", "Europe/Bucharest",
114     "America/Lower_Princes", "Atlantic", "America/Halifax",
115     "Pacific/Chuuk", "Truk", "Pacific/Truk",
116     "America/Anchorage", "Alaska", "America/Juneau",
117     "America/Rankin_Inlet", "America_Central", "America/Chicago",
118     "America/Marigot", "Atlantic", "America/Halifax",
119     "Africa/Juba", "Africa_Eastern", "Africa/Nairobi",
120     "Africa/Algiers", "Europe_Central", "Europe/Paris",
121     "Europe/Kiev", "Europe_Eastern", "Europe/Bucharest",
122     "America/Santarem", "Brasilia", "America/Sao_Paulo",
123     "Africa/Brazzaville", "Africa_Western", "Africa/Lagos",
124     "Asia/Choibalsan", "Choibalsan", "Asia/Choibalsan",
125     "Indian/Christmas", "Christmas", "Indian/Christmas",
126     "America/Nassau", "America_Eastern", "America/New_York",
127     "Africa/Tunis", "Europe_Central", "Europe/Paris",
128     "Pacific/Noumea", "New_Caledonia", "Pacific/Noumea",
129     "Africa/El_Aaiun", "Europe_Western", "Atlantic/Canary",
130     "Europe/Sarajevo", "Europe_Central", "Europe/Paris",
131     "America/Campo_Grande", "Amazon", "America/Manaus",
132     "America/Puerto_Rico", "Atlantic", "America/Halifax",
133     "Antarctica/Mawson", "Mawson", "Antarctica/Mawson",
134     "Pacific/Galapagos", "Galapagos", "Pacific/Galapagos",
135     "Asia/Tehran", "Iran", "Asia/Tehran",
136     "America/Port-au-Prince", "America_Eastern", "America/New_York",
137     "America/Scoresbysund", "Greenland_Eastern", "America/Scoresbysund",
138     "Africa/Harare", "Africa_Central", "Africa/Maputo",
139     "America/Dominica", "Atlantic", "America/Halifax",
140     "Europe/Chisinau", "Europe_Eastern", "Europe/Bucharest",
```

```

141 "America/Chihuahua", "America_Mountain", "America/Denver",
142 "America/La_Paz", "Bolivia", "America/La_Paz",
143 "Indian/Chagos", "Indian_Ocean", "Indian/Chagos",
144 "Australia/Broken_Hill", "Australia_Central", "Australia/Adelaide",
145 "America/Grenada", "Atlantic", "America/Halifax",
146 "America/North_Dakota/New_Salem", "America_Central", "America/Chicago",
147 "Pacific/Majuro", "Marshall_Islands", "Pacific/Majuro",
148 "Australia/Adelaide", "Australia_Central", "Australia/Adelaide",
149 "Europe/Warsaw", "Europe_Central", "Europe/Paris",
150 "Europe/Vienna", "Europe_Central", "Europe/Paris",
151 "Atlantic/Cape_Verde", "Cape_Verde", "Atlantic/Cape_Verde",
152 "America/Mendoza", "Argentina", "America/Buenos_Aires",
153 "Pacific/Gambier", "Gambier", "Pacific/Gambier",
154 "Europe/Istanbul", "Europe_Eastern", "Europe/Bucharest",
155 "America/Kentucky/Monticello", "America_Eastern", "America/New_York",
156 "America/Chicago", "America_Central", "America/Chicago",
157 "Asia/Ulaanbaatar", "Mongolia", "Asia/Ulaanbaatar",
158 "Indian/Maldives", "Maldives", "Indian/Maldives",
159 "America/Mexico_City", "America_Central", "America/Chicago",
160 "Africa/Asmara", "Africa_Eastern", "Africa/Nairobi",
161 "Asia/Chongqing", "China", "Asia/Shanghai",
162 "America/Argentina/La_Rioja", "Argentina", "America/Buenos_Aires",
163 "America/Tijuana", "America_Pacific", "America/Los_Angeles",
164 "Asia/Harbin", "China", "Asia/Shanghai",
165 "Pacific/Honolulu", "Hawaii_Aleutian", "Pacific/Honolulu",
166 "Atlantic/Azores", "Azores", "Atlantic/Azores",
167 "Indian/Mayotte", "Africa_Eastern", "Africa/Nairobi",
168 "America/Guatemala", "America_Central", "America/Chicago",
169 "America/Indianapolis", "America_Eastern", "America/New_York",
170 "America/Halifax", "Atlantic", "America/Halifax",
171 "America/Resolute", "America_Central", "America/Chicago",
172 "Europe/London", "GMT", "Atlantic/Reykjavik",
173 "America/Hermosillo", "America_Mountain", "America/Denver",
174 "Atlantic/Madeira", "Europe_Western", "Atlantic/Canary",
175 "Europe/Zagreb", "Europe_Central", "Europe/Paris",
176 "America/Boa_Vista", "Amazon", "America/Manaus",
177 "America/Regina", "America_Central", "America/Chicago",
178 "America/Cordoba", "Argentina", "America/Buenos_Aires",
179 "America/Shiprock", "America_Mountain", "America/Denver",
180 "Europe/Luxembourg", "Europe_Central", "Europe/Paris",
181 "America/Cancun", "America_Central", "America/Chicago",
182 "Pacific/Enderbury", "Phoenix_Islands", "Pacific/Enderbury",
183 "Africa/Bissau", "GMT", "Atlantic/Reykjavik",
184 "Antarctica/Vostok", "Vostok", "Antarctica/Vostok",
185 "Pacific/Apia", "Samoa", "Pacific/Apia",
186 "Australia/Perth", "Australia_Western", "Australia/Perth",
187 "America/Juneau", "Alaska", "America/Juneau",
188 "Africa/Mbabane", "Africa_Southern", "Africa/Johannesburg",
189 "Pacific/Niue", "Niue", "Pacific/Niue",
190 "Europe/Zurich", "Europe_Central", "Europe/Paris",
191 "America/Rio_Branco", "Amazon", "America/Manaus",
192 "Africa/Ndjamena", "Africa_Western", "Africa/Lagos",
193 "Asia/Macau", "China", "Asia/Shanghai",

```

```
194 "America/Lima", "Peru", "America/Lima",
195 "Africa/Windhoek", "Africa_Western", "Africa/Lagos",
196 "America/Sitka", "Alaska", "America/Juneau",
197 "America/Mazatlan", "America_Mountain", "America/Denver",
198 "Asia/Saigon", "Indochina", "Asia/Saigon",
199 "Asia/Kamchatka", "Magadan", "Asia/Magadan",
200 "America/Menominee", "America_Central", "America/Chicago",
201 "America/Belize", "America_Central", "America/Chicago",
202 "America/Sao_Paulo", "Brasilia", "America/Sao_Paulo",
203 "America/Barbados", "Atlantic", "America/Halifax",
204 "America/Porto_Velho", "Amazon", "America/Manaus",
205 "America/Costa_Rica", "America_Central", "America/Chicago",
206 "Europe/Monaco", "Europe_Central", "Europe/Paris",
207 "Europe/Riga", "Europe_Eastern", "Europe/Bucharest",
208 "Europe/Vatican", "Europe_Central", "Europe/Paris",
209 "Europe/Madrid", "Europe_Central", "Europe/Paris",
210 "Africa/Dakar", "GMT", "Atlantic/Reykjavik",
211 "Asia/Damascus", "Europe_Eastern", "Europe/Bucharest",
212 "Asia/Hong_Kong", "Hong_Kong", "Asia/Hong_Kong",
213 "America/Adak", "Hawaii_Aleutian", "Pacific/Honolulu",
214 "Europe/Vilnius", "Europe_Eastern", "Europe/Bucharest",
215 "America/Indiana/Indianapolis", "America_Eastern", "America/New_York",
216 "Africa/Freetown", "GMT", "Atlantic/Reykjavik",
217 "Atlantic/Reykjavik", "GMT", "Atlantic/Reykjavik",
218 "Asia/Ho_Chi_Minh", "Indochina", "Asia/Saigon",
219 "America/St_Kitts", "Atlantic", "America/Halifax",
220 "America/Martinique", "Atlantic", "America/Halifax",
221 "America/Thule", "Atlantic", "America/Halifax",
222 "America/Asuncion", "Paraguay", "America/Asuncion",
223 "Africa/Luanda", "Africa_Western", "Africa/Lagos",
224 "America/Monterrey", "America_Central", "America/Chicago",
225 "Pacific/Fiji", "Fiji", "Pacific/Fiji",
226 "Africa/Banjul", "GMT", "Atlantic/Reykjavik",
227 "America/Grand_Turk", "America_Eastern", "America/New_York",
228 "Pacific/Pitcairn", "Pitcairn", "Pacific/Pitcairn",
229 "America/Montevideo", "Uruguay", "America/Montevideo",
230 "America/Bahia_Banderas", "America_Central", "America/Chicago",
231 "America/Cayman", "America_Eastern", "America/New_York",
232 "Pacific/Norfolk", "Norfolk", "Pacific/Norfolk",
233 "Africa/Ouagadougou", "GMT", "Atlantic/Reykjavik",
234 "America/Maceio", "Brasilia", "America/Sao_Paulo",
235 "Pacific/Guam", "Chamorro", "Pacific/Saipan",
236 "Africa/Monrovia", "GMT", "Atlantic/Reykjavik",
237 "Africa/Bamako", "GMT", "Atlantic/Reykjavik",
238 "Asia/Colombo", "India", "Asia/Calcutta",
239 "Asia/Urumqi", "China", "Asia/Shanghai",
240 "Asia/Kabul", "Afghanistan", "Asia/Kabul",
241 "America/Yakutat", "Alaska", "America/Juneau",
242 "America/Phoenix", "America_Mountain", "America/Denver",
243 "Asia/Nicosia", "Europe_Eastern", "Europe/Bucharest",
244 "Asia/Phnom_Penh", "Indochina", "Asia/Saigon",
245 "America/Rainy_River", "America_Central", "America/Chicago",
246 "Europe/Uzhgorod", "Europe_Eastern", "Europe/Bucharest",
```



```

247 "Pacific/Saipan", "Chamorro", "Pacific/Saipan",
248 "America/St_Vincent", "Atlantic", "America/Halifax",
249 "Europe/Rome", "Europe_Central", "Europe/Paris",
250 "America/Nome", "Alaska", "America/Juneau",
251 "Africa/Mogadishu", "Africa_Eastern", "Africa/Nairobi",
252 "Europe/Zaporozhye", "Europe_Eastern", "Europe/Bucharest",
253 "Pacific/Funafuti", "Tuvalu", "Pacific/Funafuti",
254 "Atlantic/South_Georgia", "South_Georgia", "Atlantic/South_Georgia",
255 "Europe/Skopje", "Europe_Central", "Europe/Paris",
256 "Asia/Yekaterinburg", "Yekaterinburg", "Asia/Yekaterinburg",
257 "Australia/Melbourne", "Australia_Eastern", "Australia/Sydney",
258 "America/Argentina/Cordoba", "Argentina", "America/Buenos_Aires",
259 "Africa/Kigali", "Africa_Central", "Africa/Maputo",
260 "Africa/Blantyre", "Africa_Central", "Africa/Maputo",
261 "Africa/Tripoli", "Europe_Eastern", "Europe/Bucharest",
262 "Africa/Gaborone", "Africa_Central", "Africa/Maputo",
263 "Asia/Kuching", "Malaysia", "Asia/Kuching",
264 "Pacific/Nauru", "Nauru", "Pacific/Nauru",
265 "America/Aruba", "Atlantic", "America/Halifax",
266 "America/Antigua", "Atlantic", "America/Halifax",
267 "Europe/Volgograd", "Volgograd", "Europe/Volgograd",
268 "Africa/Djibouti", "Africa_Eastern", "Africa/Nairobi",
269 "America/Catamarca", "Argentina", "America/Buenos_Aires",
270 "Asia/Manila", "Philippines", "Asia/Manila",
271 "Pacific/Kiritimati", "Line_Islands", "Pacific/Kiritimati",
272 "Asia/Shanghai", "China", "Asia/Shanghai",
273 "Pacific/Truk", "Truk", "Pacific/Truk",
274 "Pacific/Tarawa", "Gilbert_Islands", "Pacific/Tarawa",
275 "Africa/Conakry", "GMT", "Atlantic/Reykjavik",
276 "Asia/Bishkek", "Kyrgystan", "Asia/Bishkek",
277 "Europe/Gibraltar", "Europe_Central", "Europe/Paris",
278 "Asia/Rangoon", "Myanmar", "Asia/Rangoon",
279 "Asia/Baku", "Azerbaijan", "Asia/Baku",
280 "America/Santiago", "Chile", "America/Santiago",
281 "America/El_Salvador", "America_Central", "America/Chicago",
282 "America/Noronha", "Noronha", "America/Noronha",
283 "America/St_Thomas", "Atlantic", "America/Halifax",
284 "Atlantic/St_Helena", "GMT", "Atlantic/Reykjavik",
285 "Asia/Krasnoyarsk", "Krasnoyarsk", "Asia/Krasnoyarsk",
286 "America/Vancouver", "America_Pacific", "America/Los_Angeles",
287 "Europe/Belgrade", "Europe_Central", "Europe/Paris",
288 "America/St_Barthlemy", "Atlantic", "America/Halifax",
289 "Asia/Pontianak", "Indonesia_Western", "Asia/Jakarta",
290 "Africa/Lusaka", "Africa_Central", "Africa/Maputo",
291 "America/Godthab", "Greenland_Western", "America/Godthab",
292 "Asia/Dhaka", "Bangladesh", "Asia/Dhaka",
293 "Asia/Dubai", "Gulf", "Asia/Dubai",
294 "Europe/Moscow", "Moscow", "Europe/Moscow",
295 "America/Louisville", "America_Eastern", "America/New_York",
296 "Australia/Darwin", "Australia_Central", "Australia/Adelaide",
297 "America/Santo_Domingo", "Atlantic", "America/Halifax",
298 "America/Argentina/Ushuaia", "Argentina", "America/Buenos_Aires",
299 "America/Tegucigalpa", "America_Central", "America/Chicago",

```



```
300 "Asia/Aden", "Arabian", "Asia/Riyadh",
301 "America/Inuvik", "America_Mountain", "America/Denver",
302 "Asia/Beirut", "Europe_Eastern", "Europe/Bucharest",
303 "Asia/Qatar", "Arabian", "Asia/Riyadh",
304 "Europe/Oslo", "Europe_Central", "Europe/Paris",
305 "Asia/Anadyr", "Magadan", "Asia/Magadan",
306 "Pacific/Palau", "Palau", "Pacific/Palau",
307 "Arctic/Longyearbyen", "Europe_Central", "Europe/Paris",
308 "America/Anguilla", "Atlantic", "America/Halifax",
309 "Asia/Aqttau", "Kazakhstan_Western", "Asia/Aqtobe",
310 "Asia/Yerevan", "Armenia", "Asia/Yerevan",
311 "Africa/Lagos", "Africa_Western", "Africa/Lagos",
312 "America/Denver", "America_Mountain", "America/Denver",
313 "Antarctica/Palmer", "Chile", "America/Santiago",
314 "Europe/Stockholm", "Europe_Central", "Europe/Paris",
315 "America/Bahia", "Brasilia", "America/Sao_Paulo",
316 "America/Danmarkshavn", "GMT", "Atlantic/Reykjavik",
317 "Indian/Mauritius", "Mauritius", "Indian/Mauritius",
318 "Pacific/Chatham", "Chatham", "Pacific/Chatham",
319 "Europe/Prague", "Europe_Central", "Europe/Paris",
320 "America/Blanc-Sablon", "Atlantic", "America/Halifax",
321 "America/Bogota", "Colombia", "America/Bogota",
322 "America/Managua", "America_Central", "America/Chicago",
323 "Pacific/Auckland", "New_Zealand", "Pacific/Auckland",
324 "Atlantic/Faroe", "Europe_Western", "Atlantic/Canary",
325 "America/Cambridge_Bay", "America_Mountain", "America/Denver",
326 "America/Los_Angeles", "America_Pacific", "America/Los_Angeles",
327 "Africa/Khartoum", "Africa_Eastern", "Africa/Nairobi",
328 "Europe/Simferopol", "Europe_Eastern", "Europe/Bucharest",
329 "Australia/Currie", "Australia_Eastern", "Australia/Sydney",
330 "Europe/Guernsey", "GMT", "Atlantic/Reykjavik",
331 "Asia/Thimphu", "Bhutan", "Asia/Thimphu",
332 "America/Eirunepe", "Amazon", "America/Manaus",
333 "Africa/Nairobi", "Africa_Eastern", "Africa/Nairobi",
334 "Asia/Yakutsk", "Yakutsk", "Asia/Yakutsk",
335 "Asia/Yangon", "Myanmar", "Asia/Rangoon",
336 "America/Goose_Bay", "Atlantic", "America/Halifax",
337 "Africa/Maseru", "Africa_Southern", "Africa/Johannesburg",
338 "America/Swift_Current", "America_Central", "America/Chicago",
339 "America/Guyana", "Guyana", "America/Guyana",
340 "Asia/Tokyo", "Japan", "Asia/Tokyo",
341 "Indian/Kerguelen", "French_Southern", "Indian/Kerguelen",
342 "America/Belem", "Brasilia", "America/Sao_Paulo",
343 "Pacific/Wallis", "Wallis", "Pacific/Wallis",
344 "America/Whitehorse", "America_Pacific", "America/Los_Angeles",
345 "America/North_Dakota/Beulah", "America_Central", "America/Chicago",
346 "Asia/Jerusalem", "Israel", "Asia/Jerusalem",
347 "Antarctica/Syowa", "Syowa", "Antarctica/Syowa",
348 "America/Thunder_Bay", "America_Eastern", "America/New_York",
349 "Asia/Brunei", "Brunei", "Asia/Brunei",
350 "America/Metlakatla", "America_Pacific", "America/Los_Angeles",
351 "Asia/Dushanbe", "Tajikistan", "Asia/Dushanbe",
352 "Pacific/Kosrae", "Kosrae", "Pacific/Kosrae",
```

```

353 "America/Coral_Harbour", "America_Eastern", "America/New_York",
354 "America/Tortola", "Atlantic", "America/Halifax",
355 "Asia/Karachi", "Pakistan", "Asia/Karachi",
356 "Indian/Reunion", "Reunion", "Indian/Reunion",
357 "America/Detroit", "America_Eastern", "America/New_York",
358 "Australia/Eucla", "Australia_CentralWestern", "Australia/Eucla",
359 "Asia/Seoul", "Korea", "Asia/Seoul",
360 "Asia/Singapore", "Singapore", "Asia/Singapore",
361 "Africa/Casablanca", "Europe_Western", "Atlantic/Canary",
362 "Asia/Dili", "East_Timor", "Asia/Dili",
363 "America/Indiana/Vincennes", "America_Eastern", "America/New_York",
364 "Europe/Dublin", "GMT", "Atlantic/Reykjavik",
365 "America/St_Johns", "Newfoundland", "America/St_Johns",
366 "Antarctica/Macquarie", "Macquarie", "Antarctica/Macquarie",
367 "America/Port_of_Spain", "Atlantic", "America/Halifax",
368 "Europe/Budapest", "Europe_Central", "Europe/Paris",
369 "America/Fortaleza", "Brasilia", "America/Sao_Paulo",
370 "Australia/Brisbane", "Australia_Eastern", "Australia/Sydney",
371 "Atlantic/Bermuda", "Atlantic", "America/Halifax",
372 "Asia/Amman", "Europe_Eastern", "Europe/Bucharest",
373 "Asia/Tashkent", "Uzbekistan", "Asia/Tashkent",
374 "Antarctica/DumontDURville", "DumontDURville", "Antarctica/DumontDURville",
375 "Antarctica/Casey", "Australia_Western", "Australia/Perth",
376 "Asia/Vientiane", "Indochina", "Asia/Saigon",
377 "Pacific/Johnston", "Hawaii_Aleutian", "Pacific/Honolulu",
378 "America/Jamaica", "America_Eastern", "America/New_York",
379 "Africa/Addis_Ababa", "Africa_Eastern", "Africa/Nairobi",
380 "Pacific/Ponape", "Ponape", "Pacific/Ponape",
381 "Europe/Jersey", "GMT", "Atlantic/Reykjavik",
382 "Africa/Lome", "GMT", "Atlantic/Reykjavik",
383 "America/Manaus", "Amazon", "America/Manaus",
384 "Africa/Niamey", "Africa_Western", "Africa/Lagos",
385 "Asia/Kashgar", "China", "Asia/Shanghai",
386 "Pacific/Tongatapu", "Tonga", "Pacific/Tongatapu",
387 "Europe/Minsk", "Europe_Eastern", "Europe/Bucharest",
388 "America/Edmonton", "America_Mountain", "America/Denver",
389 "Asia/Baghdad", "Arabian", "Asia/Riyadh",
390 "Asia/Kathmandu", "Nepal", "Asia/Katmandu",
391 "America/Ojinaga", "America_Mountain", "America/Denver",
392 "Africa/Abidjan", "GMT", "Atlantic/Reykjavik",
393 "America/Indiana/Winamac", "America_Eastern", "America/New_York",
394 "Asia/Qyzylorda", "Kazakhstan_Eastern", "Asia/Almaty",
395 "Australia/Sydney", "Australia_Eastern", "Australia/Sydney",
396 "Asia/Ashgabat", "Turkmenistan", "Asia/Ashgabat",
397 "Europe/Amsterdam", "Europe_Central", "Europe/Paris",
398 "America/Dawson_Creek", "America_Mountain", "America/Denver",
399 "Africa/Cairo", "Europe_Eastern", "Europe/Bucharest",
400 "Asia/Pyongyang", "Korea", "Asia/Seoul",
401 "Africa/Kampala", "Africa_Eastern", "Africa/Nairobi",
402 "America/Araguaina", "Brasilia", "America/Sao_Paulo",
403 "Asia/Novokuznetsk", "Novosibirsk", "Asia/Novosibirsk",
404 "Pacific/Kwajalein", "Marshall_Islands", "Pacific/Majuro",
405 "Africa/Lubumbashi", "Africa_Central", "Africa/Maputo",

```

```
406 "Asia/Sakhalin", "Sakhalin", "Asia/Sakhalin",
407 "America/Indiana/Vevay", "America_Eastern", "America/New_York",
408 "Africa/Maputo", "Africa_Central", "Africa/Maputo",
409 "Atlantic/Faeroe", "Europe_Western", "Atlantic/Canary",
410 "America/North_Dakota/Center", "America_Central", "America/Chicago",
411 "Pacific/Wake", "Wake", "Pacific/Wake",
412 "Pacific/Pago_Pago", "Samoa", "Pacific/Apia",
413 "America/Moncton", "Atlantic", "America/Halifax",
414 "Africa/Sao_Tome", "GMT", "Atlantic/Reykjavik",
415 "America/Glace_Bay", "Atlantic", "America/Halifax",
416 "Asia/Jakarta", "Indonesia_Western", "Asia/Jakarta",
417 "Africa/Asmera", "Africa_Eastern", "Africa/Nairobi",
418 "Europe/Lisbon", "Europe_Western", "Atlantic/Canary",
419 "America/Dawson", "America_Pacific", "America/Los_Angeles",
420 "America/Cayenne", "French_Guiana", "America/Cayenne",
421 "Asia/Bahrain", "Arabian", "Asia/Riyadh",
422 "Europe/Malta", "Europe_Central", "Europe/Paris",
423 "America/Indiana/Tell_City", "America_Central", "America/Chicago",
424 "America/Indiana/Petersburg", "America_Eastern", "America/New_York",
425 "Antarctica/Rothera", "Rothera", "Antarctica/Rothera",
426 "Asia/Aqtobe", "Kazakhstan_Western", "Asia/Aqtobe",
427 "Europe/Vaduz", "Europe_Central", "Europe/Paris",
428 "America/Indiana/Marengo", "America_Eastern", "America/New_York",
429 "Europe/Brussels", "Europe_Central", "Europe/Paris",
430 "Europe/Andorra", "Europe_Central", "Europe/Paris",
431 "America/Indiana/Knox", "America_Central", "America/Chicago",
432 "Pacific/Easter", "Easter", "Pacific/Easter",
433 "America/Argentina/Rio_Gallegos", "Argentina", "America/Buenos_Aires",
434 "Asia/Oral", "Kazakhstan_Western", "Asia/Aqtobe",
435 "Europe/Copenhagen", "Europe_Central", "Europe/Paris",
436 "Africa/Johannesburg", "Africa_Southern", "Africa/Johannesburg",
437 "Pacific/Pohnpei", "Ponape", "Pacific/Ponape",
438 "America/Argentina/Tucuman", "Argentina", "America/Buenos_Aires",
439 "America/Toronto", "America_Eastern", "America/New_York",
440 "Asia/Makassar", "Indonesia_Central", "Asia/Makassar",
441 "Europe/Berlin", "Europe_Central", "Europe/Paris",
442 "America/Argentina/Mendoza", "Argentina", "America/Buenos_Aires",
443 "America/Cuiaba", "Amazon", "America/Manaus",
444 "America/Creston", "America_Mountain", "America/Denver",
445 "Asia/Samarkand", "Uzbekistan", "Asia/Tashkent",
446 "Asia/Hovd", "Hovd", "Asia/Hovd",
447 "Europe/Bratislava", "Europe_Central", "Europe/Paris",
448 "Africa/Accra", "GMT", "Atlantic/Reykjavik",
449 "Africa/Douala", "Africa_Western", "Africa/Lagos",
450 "Africa/Nouakchott", "GMT", "Atlantic/Reykjavik",
451 "Europe/Sofia", "Europe_Eastern", "Europe/Bucharest",
452 "Antarctica/Davis", "Davis", "Antarctica/Davis",
453 "Antarctica/McMurdo", "New_Zealand", "Pacific/Auckland",
454 "Europe/San_Marino", "Europe_Central", "Europe/Paris",
455 "Africa/Porto-Novo", "Africa_Western", "Africa/Lagos",
456 "Asia/Jayapura", "Indonesia_Eastern", "Asia/Jayapura",
457 "America/St_Lucia", "Atlantic", "America/Halifax",
458 "America/Nipigon", "America_Eastern", "America/New_York",
```

```

459     "America/Argentina/Catamarca", "Argentina", "America/Buenos_Aires",
460     "Europe/Isle_of_Man", "GMT", "Atlantic/Reykjavik",
461     "America/Kentucky/Louisville", "America_Eastern", "America/New_York",
462     "America/Merida", "America_Central", "America/Chicago",
463     "Pacific/Marquesas", "Marquesas", "Pacific/Marquesas",
464     "Asia/Magadan", "Magadan", "Asia/Magadan",
465     "Africa/Libreville", "Africa_Western", "Africa/Lagos",
466     "Pacific/Efate", "Vanuatu", "Pacific/Efate",
467     "Asia/Kuala_Lumpur", "Malaysia", "Asia/Kuching",
468     "America/Iqaluit", "America_Eastern", "America/New_York",
469     "Indian/Comoro", "Africa_Eastern", "Africa/Nairobi",
470     "America/Panama", "America_Eastern", "America/New_York",
471     "Asia/Hebron", "Europe_Eastern", "Europe/Bucharest",
472     "America/Jujuy", "Argentina", "America/Buenos_Aires",
473     "America/Pangnirtung", "America_Eastern", "America/New_York",
474     "Asia/Tbilisi", "Georgia", "Asia/Tbilisi",
475     "Europe/Podgorica", "Europe_Central", "Europe/Paris",
476     "America/Boise", "America_Mountain", "America/Denver",
477     "Asia/Muscat", "Gulf", "Asia/Dubai",
478     "Indian/Mahe", "Seychelles", "Indian/Mahe",
479     "America/Montreal", "America_Eastern", "America/New_York",
480     "Africa/Bangui", "Africa_Western", "Africa/Lagos",
481     "America/Curacao", "Atlantic", "America/Halifax",
482     "Asia/Taipei", "Taipei", "Asia/Taipei",
483     "Europe/Ljubljana", "Europe_Central", "Europe/Paris",
484     "Atlantic/Stanley", "Falkland", "Atlantic/Stanley",
485     "Pacific/Guadalcanal", "Solomon", "Pacific/Guadalcanal",
486     "Asia/Kuwait", "Arabian", "Asia/Riyadh",
487     "Asia/Riyadh", "Arabian", "Asia/Riyadh",
488     "Europe/Tallinn", "Europe_Eastern", "Europe/Bucharest",
489     "America/New_York", "America_Eastern", "America/New_York",
490     "America/Paramaribo", "Suriname", "America/Paramaribo",
491     "America/Argentina/Buenos_Aires", "Argentina", "America/Buenos_Aires",
492     "Asia/Irkutsk", "Irkutsk", "Asia/Irkutsk",
493     "Asia/Katmandu", "Nepal", "Asia/Katmandu",
494     "America/Kralendijk", "Atlantic", "America/Halifax",
495 };
496 private static final String[] mzoneMap = new String[] {
497     "GMT", "ST", "Africa/Sao_Tome",
498     "GMT", "ML", "Africa/Bamako",
499     "GMT", "IE", "Europe/Dublin",
500     "GMT", "SN", "Africa/Dakar",
501     "GMT", "GH", "Africa/Accra",
502     "GMT", "CI", "Africa/Abidjan",
503     "GMT", "BF", "Africa/Ouagadougou",
504     "GMT", "MR", "Africa/Nouakchott",
505     "GMT", "GM", "Africa/Banjul",
506     "GMT", "SL", "Africa/Freetown",
507     "GMT", "GN", "Africa/Conakry",
508     "GMT", "SH", "Atlantic/St_Helena",
509     "GMT", "GB", "Europe/London",
510     "GMT", "LR", "Africa/Monrovia",
511     "GMT", "TG", "Africa/Lome",

```

```
512 "Africa_Western", "CF", "Africa/Bangui",
513 "Africa_Western", "NE", "Africa/Niamey",
514 "Africa_Western", "CM", "Africa/Douala",
515 "Africa_Western", "CD", "Africa/Kinshasa",
516 "Africa_Western", "CG", "Africa/Brazzaville",
517 "Africa_Western", "GA", "Africa/Libreville",
518 "Africa_Western", "TD", "Africa/Ndjamena",
519 "Africa_Western", "AO", "Africa/Luanda",
520 "Africa_Western", "GQ", "Africa/Malabo",
521 "Africa_Eastern", "YT", "Indian/Mayotte",
522 "Africa_Eastern", "UG", "Africa/Kampala",
523 "Africa_Eastern", "ET", "Africa/Addis_Ababa",
524 "Africa_Eastern", "MG", "Indian/Antananarivo",
525 "Africa_Eastern", "TZ", "Africa/Dar_es_Salaam",
526 "Africa_Eastern", "SO", "Africa/Mogadishu",
527 "Africa_Eastern", "ER", "Africa/Asmera",
528 "Africa_Eastern", "KM", "Indian/Comoro",
529 "Africa_Eastern", "DJ", "Africa/Djibouti",
530 "Europe_Central", "GI", "Europe/Gibraltar",
531 "Europe_Central", "DK", "Europe/Copenhagen",
532 "Europe_Central", "SE", "Europe/Stockholm",
533 "Europe_Central", "CH", "Europe/Zurich",
534 "Europe_Central", "AL", "Europe/Tirane",
535 "Europe_Central", "RS", "Europe/Belgrade",
536 "Europe_Central", "HU", "Europe/Budapest",
537 "Europe_Central", "MT", "Europe/Malta",
538 "Europe_Central", "PL", "Europe/Warsaw",
539 "Europe_Central", "ME", "Europe/Podgorica",
540 "Europe_Central", "ES", "Europe/Madrid",
541 "Europe_Central", "CZ", "Europe/Prague",
542 "Europe_Central", "IT", "Europe/Rome",
543 "Europe_Central", "SI", "Europe/Ljubljana",
544 "Europe_Central", "LI", "Europe/Vaduz",
545 "Europe_Central", "AT", "Europe/Vienna",
546 "Europe_Central", "VA", "Europe/Vatican",
547 "Europe_Central", "DE", "Europe/Berlin",
548 "Europe_Central", "NO", "Europe/Oslo",
549 "Europe_Central", "SK", "Europe/Bratislava",
550 "Europe_Central", "AD", "Europe/Andorra",
551 "Europe_Central", "SM", "Europe/San_Marino",
552 "Europe_Central", "MK", "Europe/Skopje",
553 "Europe_Central", "TN", "Africa/Tunis",
554 "Europe_Central", "HR", "Europe/Zagreb",
555 "Europe_Central", "NL", "Europe/Amsterdam",
556 "Europe_Central", "BE", "Europe/Brussels",
557 "Europe_Central", "MC", "Europe/Monaco",
558 "Europe_Central", "LU", "Europe/Luxembourg",
559 "Europe_Central", "BA", "Europe/Sarajevo",
560 "China", "MO", "Asia/Macau",
561 "America_Pacific", "MX", "America/Tijuana",
562 "America_Pacific", "CA", "America/Vancouver",
563 "Indochina", "LA", "Asia/Vientiane",
564 "Indochina", "KH", "Asia/Phnom_Penh",
```

```
565 "Indochina", "TH", "Asia/Bangkok",
566 "Korea", "KP", "Asia/Pyongyang",
567 "America_Mountain", "MX", "America/Hermosillo",
568 "America_Mountain", "CA", "America/Edmonton",
569 "Africa_Southern", "LS", "Africa/Maseru",
570 "Africa_Southern", "SZ", "Africa/Mbabane",
571 "Chile", "AQ", "Antarctica/Palmer",
572 "New_Zealand", "AQ", "Antarctica/McMurdo",
573 "Gulf", "OM", "Asia/Muscat",
574 "Europe_Western", "FO", "Atlantic/Faeroe",
575 "America_Eastern", "TC", "America/Grand_Turk",
576 "America_Eastern", "CA", "America/Toronto",
577 "America_Eastern", "BS", "America/Nassau",
578 "America_Eastern", "PA", "America/Panama",
579 "America_Eastern", "JM", "America/Jamaica",
580 "America_Eastern", "KY", "America/Cayman",
581 "Africa_Central", "BI", "Africa/Bujumbura",
582 "Africa_Central", "ZM", "Africa/Lusaka",
583 "Africa_Central", "ZW", "Africa/Harare",
584 "Africa_Central", "CD", "Africa/Lubumbashi",
585 "Africa_Central", "BW", "Africa/Gaborone",
586 "Africa_Central", "RW", "Africa/Kigali",
587 "Africa_Central", "MW", "Africa/Blantyre",
588 "America_Central", "MX", "America/Mexico_City",
589 "America_Central", "HN", "America/Tegucigalpa",
590 "America_Central", "CA", "America/Winnipeg",
591 "America_Central", "GT", "America/Guatemala",
592 "America_Central", "SV", "America/El_Salvador",
593 "America_Central", "CR", "America/Costa_Rica",
594 "America_Central", "BZ", "America/Belize",
595 "Atlantic", "MS", "America/Montserrat",
596 "Atlantic", "AG", "America/Antigua",
597 "Atlantic", "TT", "America/Port_of_Spain",
598 "Atlantic", "MQ", "America/Martinique",
599 "Atlantic", "DM", "America/Dominica",
600 "Atlantic", "KN", "America/St_Kitts",
601 "Atlantic", "BM", "Atlantic/Bermuda",
602 "Atlantic", "PR", "America/Puerto_Rico",
603 "Atlantic", "AW", "America/Aruba",
604 "Atlantic", "VG", "America/Tortola",
605 "Atlantic", "GD", "America/Grenada",
606 "Atlantic", "GL", "America/Thule",
607 "Atlantic", "BB", "America/Barbados",
608 "Atlantic", "BQ", "America/Kralendijk",
609 "Atlantic", "SX", "America/Lower_Princes",
610 "Atlantic", "VI", "America/St_Thomas",
611 "Atlantic", "MF", "America/Marigot",
612 "Atlantic", "AI", "America/Anguilla",
613 "Atlantic", "AN", "America/Curacao",
614 "Atlantic", "LC", "America/St_Lucia",
615 "Atlantic", "GP", "America/Guadeloupe",
616 "Atlantic", "VC", "America/St_Vincent",
617 "Arabian", "QA", "Asia/Qatar",
```

```

618     "Arabian", "YE", "Asia/Aden",
619     "Arabian", "KW", "Asia/Kuwait",
620     "Arabian", "BH", "Asia/Bahrain",
621     "Arabian", "IQ", "Asia/Baghdad",
622     "India", "LK", "Asia/Colombo",
623     "Europe_Eastern", "SY", "Asia/Damascus",
624     "Europe_Eastern", "BG", "Europe/Sofia",
625     "Europe_Eastern", "GR", "Europe/Athens",
626     "Europe_Eastern", "JO", "Asia/Amman",
627     "Europe_Eastern", "CY", "Asia/Nicosia",
628     "Europe_Eastern", "AX", "Europe/Mariehamn",
629     "Europe_Eastern", "LB", "Asia/Beirut",
630     "Europe_Eastern", "FI", "Europe/Helsinki",
631     "Europe_Eastern", "EG", "Africa/Cairo",
632     "Chamorro", "GU", "Pacific/Guam",
633 };
634 private static final String[] aliasMap = new String[] {
635     "Brazil/Acre", "America/Rio_Branco",
636     "US/Indiana-Starke", "America/Indiana/Knox",
637     "America/Atka", "America/Adak",
638     "America/St_Barthelemy", "America/Guadeloupe",
639     "Australia/North", "Australia/Darwin",
640     "Europe/Zagreb", "Europe/Belgrade",
641     "Etc/Universal", "Etc/UTC",
642     "NZ-CHAT", "Pacific/Chatham",
643     "Asia/Macao", "Asia/Macau",
644     "Pacific/Yap", "Pacific/Chuuk",
645     "Egypt", "Africa/Cairo",
646     "US/Central", "America/Chicago",
647     "Canada/Atlantic", "America/Halifax",
648     "Brazil/East", "America/Sao_Paulo",
649     "America/Cordoba", "America/Argentina/Cordoba",
650     "US/Hawaii", "Pacific/Honolulu",
651     "America/Louisville", "America/Kentucky/Louisville",
652     "America/Shiprock", "America/Denver",
653     "Australia/Canberra", "Australia/Sydney",
654     "Asia/Chungking", "Asia/Chongqing",
655     "Universal", "Etc/UTC",
656     "US/Alaska", "America/Anchorage",
657     "Asia/Ujung_Pandang", "Asia/Makassar",
658     "Japan", "Asia/Tokyo",
659     "Atlantic/Faeroe", "Atlantic/Faroe",
660     "Asia/Istanbul", "Europe/Istanbul",
661     "US/Pacific", "America/Los_Angeles",
662     "Mexico/General", "America/Mexico_City",
663     "Poland", "Europe/Warsaw",
664     "Africa/Asmera", "Africa/Asmara",
665     "Asia/Saigon", "Asia/Ho_Chi_Minh",
666     "US/Michigan", "America/Detroit",
667     "America/Argentina/ComodRivadavia", "America/Argentina/Catamarca",
668     "W-SU", "Europe/Moscow",
669     "Australia/ACT", "Australia/Sydney",
670     "Asia/Calcutta", "Asia/Kolkata",

```



```
671 "Arctic/Longyearbyen", "Europe/Oslo",
672 "America/Knox_IN", "America/Indiana/Knox",
673 "ROC", "Asia/Taipei",
674 "Zulu", "Etc/UTC",
675 "Australia/Yancowinna", "Australia/Broken_Hill",
676 "Australia/West", "Australia/Perth",
677 "Singapore", "Asia/Singapore",
678 "Europe/Mariehamn", "Europe/Helsinki",
679 "ROK", "Asia/Seoul",
680 "America/Porto_Acre", "America/Rio_Branco",
681 "Etc/Zulu", "Etc/UTC",
682 "Canada/Yukon", "America/Whitehorse",
683 "Europe/Vatican", "Europe/Rome",
684 "Africa/Timbuktu", "Africa/Bamako",
685 "America/Buenos_Aires", "America/Argentina/Buenos_Aires",
686 "Canada/Pacific", "America/Vancouver",
687 "US/Pacific-New", "America/Los_Angeles",
688 "Mexico/BajaNorte", "America/Tijuana",
689 "Europe/Guernsey", "Europe/London",
690 "Asia/Tel_Aviv", "Asia/Jerusalem",
691 "Chile/Continental", "America/Santiago",
692 "Jamaica", "America/Jamaica",
693 "Mexico/BajaSur", "America/Mazatlan",
694 "Canada/Eastern", "America/Toronto",
695 "Australia/Tasmania", "Australia/Hobart",
696 "NZ", "Pacific/Auckland",
697 "America/Lower_Princes", "America/Curacao",
698 "GMT-", "Etc/GMT",
699 "America/Rosario", "America/Argentina/Cordoba",
700 "Libya", "Africa/Tripoli",
701 "Asia/Ashkhabad", "Asia/Ashgabat",
702 "Australia/NSW", "Australia/Sydney",
703 "America/Marigot", "America/Guadeloupe",
704 "Europe/Bratislava", "Europe/Prague",
705 "Portugal", "Europe/Lisbon",
706 "Etc/GMT-", "Etc/GMT",
707 "Europe/San_Marino", "Europe/Rome",
708 "Europe/Sarajevo", "Europe/Belgrade",
709 "Antarctica/South_Pole", "Antarctica/McMurdo",
710 "Canada/Central", "America/Winnipeg",
711 "Etc/GMT", "Etc/GMT",
712 "Europe/Isle_of_Man", "Europe/London",
713 "America/Fort_Wayne", "America/Indiana/Indianapolis",
714 "Eire", "Europe/Dublin",
715 "America/Coral_Harbour", "America/Atikokan",
716 "Europe/Nicosia", "Asia/Nicosia",
717 "US/Samoa", "Pacific/Pago_Pago",
718 "Hongkong", "Asia/Hong_Kong",
719 "Canada/Saskatchewan", "America/Regina",
720 "Asia/Thimbu", "Asia/Thimphu",
721 "Kwajalein", "Pacific/Kwajalein",
722 "GB", "Europe/London",
723 "Chile/EasterIsland", "Pacific/Easter",
```



```

724     "US/East-Indiana", "America/Indiana/Indianapolis",
725     "Australia/LHI", "Australia/Lord_Howe",
726     "Cuba", "America/Havana",
727     "America/Jujuy", "America/Argentina/Jujuy",
728     "US/Mountain", "America/Denver",
729     "Atlantic/Jan_Mayen", "Europe/Oslo",
730     "Europe/Tiraspol", "Europe/Chisinau",
731     "Europe/Podgorica", "Europe/Belgrade",
732     "US/Arizona", "America/Phoenix",
733     "Navajo", "America/Denver",
734     "Etc/Greenwich", "Etc/GMT",
735     "Canada/Mountain", "America/Edmonton",
736     "Iceland", "Atlantic/Reykjavik",
737     "Australia/Victoria", "Australia/Melbourne",
738     "Australia/South", "Australia/Adelaide",
739     "Brazil/West", "America/Manaus",
740     "Pacific/Ponape", "Pacific/Pohnpei",
741     "Europe/Ljubljana", "Europe/Belgrade",
742     "Europe/Jersey", "Europe/London",
743     "Australia/Queensland", "Australia/Brisbane",
744     "UTC", "Etc/UTC",
745     "Canada/Newfoundland", "America/St_Johns",
746     "Europe/Skopje", "Europe/Belgrade",
747     "Canada/East-Saskatchewan", "America/Regina",
748     "PRC", "Asia/Shanghai",
749     "UCT", "Etc/UCT",
750     "America/Mendoza", "America/Argentina/Mendoza",
751     "Israel", "Asia/Jerusalem",
752     "US/Eastern", "America/New_York",
753     "Asia/Ulan_Bator", "Asia/Ulaanbaatar",
754     "Turkey", "Europe/Istanbul",
755     "GMT", "Etc/GMT",
756     "US/Aleutian", "America/Adak",
757     "Brazil/DeNoronha", "America/Noronha",
758     "GB-Eire", "Europe/London",
759     "Asia/Dacca", "Asia/Dhaka",
760     "America/Ensenada", "America/Tijuana",
761     "America/Catamarca", "America/Argentina/Catamarca",
762     "Iran", "Asia/Tehran",
763     "Greenwich", "Etc/GMT",
764     "Pacific/Truk", "Pacific/Chuuk",
765     "Pacific/Samoa", "Pacific/Pago_Pago",
766     "America/Virgin", "America/St_Thomas",
767     "Asia/Katmandu", "Asia/Kathmandu",
768     "America/Indianapolis", "America/Indiana/Indianapolis",
769     "Europe/Belfast", "Europe/London",
770     "America/Kralendijk", "America/Curacao",
771     "Asia/Rangoon", "Asia/Yangon",
772 };
773
774 private static final Map<String, String> zidToMzone = new HashMap<>();
775 private static final Map<String, String> mzoneToZid = new HashMap<>();
776 private static final Map<String, Map<String, String>> mzoneToZidL = new HashMap<>();

```

```

777     private static final Map<String, String> aliases = new HashMap<>();
778
779     static {
780         for (int i = 0; i < zidMap.length; i += 3) {
781             zidToMzone.put(zidMap[i], zidMap[i + 1]);
782             mzoneToZid.put(zidMap[i + 1], zidMap[i + 2]);
783         }
784
785         for (int i = 0; i < mzoneMap.length; i += 3) {
786             String mzone = mzoneMap[i];
787             Map<String, String> map = mzoneToZidL.get(mzone);
788             if (map == null) {
789                 map = new HashMap<>();
790                 mzoneToZidL.put(mzone, map);
791             }
792             map.put(mzoneMap[i + 1], mzoneMap[i + 2]);
793         }
794
795         for (int i = 0; i < aliasMap.length; i += 2) {
796             aliases.put(aliasMap[i], aliasMap[i + 1]);
797         }
798     }
799 }

```

6.14 package-info.java

Secuencia 111: java/time/format/package-info.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*

```

```
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
62
63 /**
64 * <p>
65 * Provides classes to print and parse dates and times.
66 * </p>
67 * <p>
68 * Printing and parsing is based around the
69 * {@link java.time.format.DateTimeFormatter DateTimeFormatter} class.
70 * Instances are generally obtained from
71 * {@link java.time.format.DateTimeFormatter DateTimeFormatter}, however
72 * {@link java.time.format.DateTimeFormatterBuilder DateTimeFormatterBuilder}
73 * can be used if more power is needed.
74 * </p>
75 * <p>
76 * Localization occurs by calling
77 * {@link java.time.format.DateTimeFormatter#withLocale(java.util.Locale) withLocale(Locale)}
78 * on the formatter. Further customization is possible using
79 * {@link java.time.format.DecimalStyle DecimalStyle}.
```

```

80 * </p>
81 *
82 * <h3>Package specification</h3>
83 * <p>
84 * Unless otherwise noted, passing a null argument to a constructor or method in any class or
      interface
85 * in this package will cause a {@link java.lang.NullPointerException NullPointerException} to be
      thrown.
86 * The Javadoc "@param" definition is used to summarise the null-behavior.
87 * The "@throws {@link java.lang.NullPointerException}" is not explicitly documented in each method
      .
88 * </p>
89 * <p>
90 * All calculations should check for numeric overflow and throw either an {@link java.lang.
      ArithmeticException}
91 * or a {@link java.time.DateTimeException}.
92 * </p>
93 * @since JDK1.8
94 */
95 package java.time.format;

```

7 Time Temporal (java.time.temporal package)

7.1 ChronoField.java

Secuencia 112: java/time/temporal/ChronoField.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos

```

```
28  *
29  * All rights reserved.
30  *
31  * Redistribution and use in source and binary forms, with or without
32  * modification, are permitted provided that the following conditions are met:
33  *
34  * * Redistributions of source code must retain the above copyright notice,
35  *   this list of conditions and the following disclaimer.
36  *
37  * * Redistributions in binary form must reproduce the above copyright notice,
38  *   this list of conditions and the following disclaimer in the documentation
39  *   and/or other materials provided with the distribution.
40  *
41  * * Neither the name of JSR-310 nor the names of its contributors
42  *   may be used to endorse or promote products derived from this software
43  *   without specific prior written permission.
44  *
45  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56  */
57 package java.time.temporal;
58
59 import static java.time.temporal.ChronoUnit.DAYS;
60 import static java.time.temporal.ChronoUnit.ERAS;
61 import static java.time.temporal.ChronoUnit.FOREVER;
62 import static java.time.temporal.ChronoUnit.HALF_DAYS;
63 import static java.time.temporal.ChronoUnit.HOURS;
64 import static java.time.temporal.ChronoUnit.MICROS;
65 import static java.time.temporal.ChronoUnit.MILLIS;
66 import static java.time.temporal.ChronoUnit.MINUTES;
67 import static java.time.temporal.ChronoUnit.MONTHS;
68 import static java.time.temporal.ChronoUnit.NANOS;
69 import static java.time.temporal.ChronoUnit.SECONDS;
70 import static java.time.temporal.ChronoUnit.WEEKS;
71 import static java.time.temporal.ChronoUnit.YEARS;
72
73 import java.time.DayOfWeek;
74 import java.time.Instant;
75 import java.time.Year;
76 import java.time.ZoneOffset;
77 import java.time.chrono.ChronoLocalDate;
78 import java.time.chrono.Chronology;
79 import java.util.Locale;
80 import java.util.Objects;
```

```

81 import java.util.ResourceBundle;
82 import sun.util.locale.provider.LocaleProviderAdapter;
83 import sun.util.locale.provider.LocaleResources;
84
85 /**
86  * A standard set of fields.
87  * <p>
88  * This set of fields provide field-based access to manipulate a date, time or date-time.
89  * The standard set of fields can be extended by implementing {@link TemporalField}.
90  * <p>
91  * These fields are intended to be applicable in multiple calendar systems.
92  * For example, most non-ISO calendar systems define dates as a year, month and day,
93  * just with slightly different rules.
94  * The documentation of each field explains how it operates.
95  *
96  * @implSpec
97  * This is a final, immutable and thread-safe enum.
98  *
99  * @since 1.8
100 */
101 public enum ChronoField implements TemporalField {
102
103     /**
104      * The nano-of-second.
105      * <p>
106      * This counts the nanosecond within the second, from 0 to 999,999,999.
107      * This field has the same meaning for all calendar systems.
108      * <p>
109      * This field is used to represent the nano-of-second handling any fraction of the second.
110      * Implementations of {@code TemporalAccessor} should provide a value for this field if
111      * they can return a value for {@link #SECOND_OF_MINUTE}, {@link #SECOND_OF_DAY} or
112      * {@link #INSTANT_SECONDS} filling unknown precision with zero.
113      * <p>
114      * When this field is used for setting a value, it should set as much precision as the
115      * object stores, using integer division to remove excess precision.
116      * For example, if the {@code TemporalAccessor} stores time to millisecond precision,
117      * then the nano-of-second must be divided by 1,000,000 before replacing the milli-of-second.
118      * <p>
119      * When parsing this field it behaves equivalent to the following:
120      * The value is validated in strict and smart mode but not in lenient mode.
121      * The field is resolved in combination with {@code MILLI_OF_SECOND} and {@code MICRO_OF_SECOND}
122      * }.
123     */
124     NANO_OF_SECOND("NanoOfSecond", NANOS, SECONDS, ValueRange.of(0, 999_999_999)),
125
126     /**
127      * The nano-of-day.
128      * <p>
129      * This counts the nanosecond within the day, from 0 to (24 * 60 * 60 * 1,000,000,000) - 1.
130      * This field has the same meaning for all calendar systems.
131      * <p>
132      * This field is used to represent the nano-of-day handling any fraction of the second.
133      * Implementations of {@code TemporalAccessor} should provide a value for this field if
134      * they can return a value for {@link #SECOND_OF_DAY} filling unknown precision with zero.

```

```

133 * <p>
134 * When parsing this field it behaves equivalent to the following:
135 * The value is validated in strict and smart mode but not in lenient mode.
136 * The value is split to form {@code NANO_OF_SECOND}, {@code SECOND_OF_MINUTE},
137 * {@code MINUTE_OF_HOUR} and {@code HOUR_OF_DAY} fields.
138 */
139 NANO_OF_DAY("NanoOfDay", NANOS, DAYS, ValueRange.of(0, 86400L * 1000_000_000L - 1)),
140 /**
141 * The micro-of-second.
142 * <p>
143 * This counts the microsecond within the second, from 0 to 999,999.
144 * This field has the same meaning for all calendar systems.
145 * <p>
146 * This field is used to represent the micro-of-second handling any fraction of the second.
147 * Implementations of {@code TemporalAccessor} should provide a value for this field if
148 * they can return a value for {@link #SECOND_OF_MINUTE}, {@link #SECOND_OF_DAY} or
149 * {@link #INSTANT_SECONDS} filling unknown precision with zero.
150 * <p>
151 * When this field is used for setting a value, it should behave in the same way as
152 * setting {@link #NANO_OF_SECOND} with the value multiplied by 1,000.
153 * <p>
154 * When parsing this field it behaves equivalent to the following:
155 * The value is validated in strict and smart mode but not in lenient mode.
156 * The field is resolved in combination with {@code MILLI_OF_SECOND} to produce
157 * {@code NANO_OF_SECOND}.
158 */
159 MICRO_OF_SECOND("MicroOfSecond", MICROS, SECONDS, ValueRange.of(0, 999_999)),
160 /**
161 * The micro-of-day.
162 * <p>
163 * This counts the microsecond within the day, from 0 to (24 * 60 * 60 * 1,000,000) - 1.
164 * This field has the same meaning for all calendar systems.
165 * <p>
166 * This field is used to represent the micro-of-day handling any fraction of the second.
167 * Implementations of {@code TemporalAccessor} should provide a value for this field if
168 * they can return a value for {@link #SECOND_OF_DAY} filling unknown precision with zero.
169 * <p>
170 * When this field is used for setting a value, it should behave in the same way as
171 * setting {@link #NANO_OF_DAY} with the value multiplied by 1,000.
172 * <p>
173 * When parsing this field it behaves equivalent to the following:
174 * The value is validated in strict and smart mode but not in lenient mode.
175 * The value is split to form {@code MICRO_OF_SECOND}, {@code SECOND_OF_MINUTE},
176 * {@code MINUTE_OF_HOUR} and {@code HOUR_OF_DAY} fields.
177 */
178 MICRO_OF_DAY("MicroOfDay", MICROS, DAYS, ValueRange.of(0, 86400L * 1000_000L - 1)),
179 /**
180 * The milli-of-second.
181 * <p>
182 * This counts the millisecond within the second, from 0 to 999.
183 * This field has the same meaning for all calendar systems.
184 * <p>
185 * This field is used to represent the milli-of-second handling any fraction of the second.

```

```

186 * Implementations of {@code TemporalAccessor} should provide a value for this field if
187 * they can return a value for {@link #SECOND_OF_MINUTE}, {@link #SECOND_OF_DAY} or
188 * {@link #INSTANT_SECONDS} filling unknown precision with zero.
189 * <p>
190 * When this field is used for setting a value, it should behave in the same way as
191 * setting {@link #NANO_OF_SECOND} with the value multiplied by 1,000,000.
192 * <p>
193 * When parsing this field it behaves equivalent to the following:
194 * The value is validated in strict and smart mode but not in lenient mode.
195 * The field is resolved in combination with {@code MICRO_OF_SECOND} to produce
196 * {@code NANO_OF_SECOND}.
197 */
198 MILLI_OF_SECOND("MilliOfSecond", MILLIS, SECONDS, ValueRange.of(0, 999)),
199 /**
200 * The milli-of-day.
201 * <p>
202 * This counts the millisecond within the day, from 0 to (24 * 60 * 60 * 1,000) - 1.
203 * This field has the same meaning for all calendar systems.
204 * <p>
205 * This field is used to represent the milli-of-day handling any fraction of the second.
206 * Implementations of {@code TemporalAccessor} should provide a value for this field if
207 * they can return a value for {@link #SECOND_OF_DAY} filling unknown precision with zero.
208 * <p>
209 * When this field is used for setting a value, it should behave in the same way as
210 * setting {@link #NANO_OF_DAY} with the value multiplied by 1,000,000.
211 * <p>
212 * When parsing this field it behaves equivalent to the following:
213 * The value is validated in strict and smart mode but not in lenient mode.
214 * The value is split to form {@code MILLI_OF_SECOND}, {@code SECOND_OF_MINUTE},
215 * {@code MINUTE_OF_HOUR} and {@code HOUR_OF_DAY} fields.
216 */
217 MILLI_OF_DAY("MilliOfDay", MILLIS, DAYS, ValueRange.of(0, 86400L * 1000L - 1)),
218 /**
219 * The second-of-minute.
220 * <p>
221 * This counts the second within the minute, from 0 to 59.
222 * This field has the same meaning for all calendar systems.
223 * <p>
224 * When parsing this field it behaves equivalent to the following:
225 * The value is validated in strict and smart mode but not in lenient mode.
226 */
227 SECOND_OF_MINUTE("SecondOfMinute", SECONDS, MINUTES, ValueRange.of(0, 59), "second"),
228 /**
229 * The second-of-day.
230 * <p>
231 * This counts the second within the day, from 0 to (24 * 60 * 60) - 1.
232 * This field has the same meaning for all calendar systems.
233 * <p>
234 * When parsing this field it behaves equivalent to the following:
235 * The value is validated in strict and smart mode but not in lenient mode.
236 * The value is split to form {@code SECOND_OF_MINUTE}, {@code MINUTE_OF_HOUR}
237 * and {@code HOUR_OF_DAY} fields.
238 */

```



```

239 SECOND_OF_DAY("SecondOfDay", SECONDS, DAYS, ValueRange.of(0, 86400L - 1)),
240 /**
241  * The minute-of-hour.
242  * <p>
243  * This counts the minute within the hour, from 0 to 59.
244  * This field has the same meaning for all calendar systems.
245  * <p>
246  * When parsing this field it behaves equivalent to the following:
247  * The value is validated in strict and smart mode but not in lenient mode.
248  */
249 MINUTE_OF_HOUR("MinuteOfHour", MINUTES, HOURS, ValueRange.of(0, 59), "minute"),
250 /**
251  * The minute-of-day.
252  * <p>
253  * This counts the minute within the day, from 0 to (24 * 60) - 1.
254  * This field has the same meaning for all calendar systems.
255  * <p>
256  * When parsing this field it behaves equivalent to the following:
257  * The value is validated in strict and smart mode but not in lenient mode.
258  * The value is split to form {@code MINUTE_OF_HOUR} and {@code HOUR_OF_DAY} fields.
259  */
260 MINUTE_OF_DAY("MinuteOfDay", MINUTES, DAYS, ValueRange.of(0, (24 * 60) - 1)),
261 /**
262  * The hour-of-am-pm.
263  * <p>
264  * This counts the hour within the AM/PM, from 0 to 11.
265  * This is the hour that would be observed on a standard 12-hour digital clock.
266  * This field has the same meaning for all calendar systems.
267  * <p>
268  * When parsing this field it behaves equivalent to the following:
269  * The value is validated from 0 to 11 in strict and smart mode.
270  * In lenient mode the value is not validated. It is combined with
271  * {@code AMPM_OF_DAY} to form {@code HOUR_OF_DAY} by multiplying
272  * the {@code AMPM_OF_DAY} value by 12.
273  */
274 HOUR_OF_AMPM("HourOfAmPm", HOURS, HALF_DAYS, ValueRange.of(0, 11)),
275 /**
276  * The clock-hour-of-am-pm.
277  * <p>
278  * This counts the hour within the AM/PM, from 1 to 12.
279  * This is the hour that would be observed on a standard 12-hour analog wall clock.
280  * This field has the same meaning for all calendar systems.
281  * <p>
282  * When parsing this field it behaves equivalent to the following:
283  * The value is validated from 1 to 12 in strict mode and from
284  * 0 to 12 in smart mode. In lenient mode the value is not validated.
285  * The field is converted to an {@code HOUR_OF_AMPM} with the same value,
286  * unless the value is 12, in which case it is converted to 0.
287  */
288 CLOCK_HOUR_OF_AMPM("ClockHourOfAmPm", HOURS, HALF_DAYS, ValueRange.of(1, 12)),
289 /**
290  * The hour-of-day.
291  * <p>

```

```

292     * This counts the hour within the day, from 0 to 23.
293     * This is the hour that would be observed on a standard 24-hour digital clock.
294     * This field has the same meaning for all calendar systems.
295     * <p>
296     * When parsing this field it behaves equivalent to the following:
297     * The value is validated in strict and smart mode but not in lenient mode.
298     * The field is combined with {@code MINUTE_OF_HOUR}, {@code SECOND_OF_MINUTE} and
299     * {@code NANO_OF_SECOND} to produce a {@code LocalDateTime}.
300     * In lenient mode, any excess days are added to the parsed date, or
301     * made available via {@link java.time.format.DateTimeFormatter#parsedExcessDays()}.
302     */
303     HOUR_OF_DAY("HourOfDay", HOURS, DAYS, ValueRange.of(0, 23), "hour"),
304     /**
305     * The clock-hour-of-day.
306     * <p>
307     * This counts the hour within the AM/PM, from 1 to 24.
308     * This is the hour that would be observed on a 24-hour analog wall clock.
309     * This field has the same meaning for all calendar systems.
310     * <p>
311     * When parsing this field it behaves equivalent to the following:
312     * The value is validated from 1 to 24 in strict mode and from
313     * 0 to 24 in smart mode. In lenient mode the value is not validated.
314     * The field is converted to an {@code HOUR_OF_DAY} with the same value,
315     * unless the value is 24, in which case it is converted to 0.
316     */
317     CLOCK_HOUR_OF_DAY("ClockHourOfDay", HOURS, DAYS, ValueRange.of(1, 24)),
318     /**
319     * The am-pm-of-day.
320     * <p>
321     * This counts the AM/PM within the day, from 0 (AM) to 1 (PM).
322     * This field has the same meaning for all calendar systems.
323     * <p>
324     * When parsing this field it behaves equivalent to the following:
325     * The value is validated from 0 to 1 in strict and smart mode.
326     * In lenient mode the value is not validated. It is combined with
327     * {@code HOUR_OF_AMPM} to form {@code HOUR_OF_DAY} by multiplying
328     * the {@code AMPM_OF_DAY} value by 12.
329     */
330     AMPM_OF_DAY("AmPmOfDay", HALF_DAYS, DAYS, ValueRange.of(0, 1), "dayperiod"),
331     /**
332     * The day-of-week, such as Tuesday.
333     * <p>
334     * This represents the standard concept of the day of the week.
335     * In the default ISO calendar system, this has values from Monday (1) to Sunday (7).
336     * The {@link DayOfWeek} class can be used to interpret the result.
337     * <p>
338     * Most non-ISO calendar systems also define a seven day week that aligns with ISO.
339     * Those calendar systems must also use the same numbering system, from Monday (1) to
340     * Sunday (7), which allows {@code DayOfWeek} to be used.
341     * <p>
342     * Calendar systems that do not have a standard seven day week should implement this field
343     * if they have a similar concept of named or numbered days within a period similar
344     * to a week. It is recommended that the numbering starts from 1.

```

```

345     */
346     DAY_OF_WEEK("DayOfWeek", DAYS, WEEKS, ValueRange.of(1, 7), "weekday"),
347     /**
348      * The aligned day-of-week within a month.
349      * <p>
350      * This represents concept of the count of days within the period of a week
351      * where the weeks are aligned to the start of the month.
352      * This field is typically used with {@link #ALIGNED_WEEK_OF_MONTH}.
353      * <p>
354      * For example, in a calendar systems with a seven day week, the first aligned-week-of-month
355      * starts on day-of-month 1, the second aligned-week starts on day-of-month 8, and so on.
356      * Within each of these aligned-weeks, the days are numbered from 1 to 7 and returned
357      * as the value of this field.
358      * As such, day-of-month 1 to 7 will have aligned-day-of-week values from 1 to 7.
359      * And day-of-month 8 to 14 will repeat this with aligned-day-of-week values from 1 to 7.
360      * <p>
361      * Calendar systems that do not have a seven day week should typically implement this
362      * field in the same way, but using the alternate week length.
363      */
364     ALIGNED_DAY_OF_WEEK_IN_MONTH("AlignedDayOfWeekInMonth", DAYS, WEEKS, ValueRange.of(1, 7)),
365     /**
366      * The aligned day-of-week within a year.
367      * <p>
368      * This represents concept of the count of days within the period of a week
369      * where the weeks are aligned to the start of the year.
370      * This field is typically used with {@link #ALIGNED_WEEK_OF_YEAR}.
371      * <p>
372      * For example, in a calendar systems with a seven day week, the first aligned-week-of-year
373      * starts on day-of-year 1, the second aligned-week starts on day-of-year 8, and so on.
374      * Within each of these aligned-weeks, the days are numbered from 1 to 7 and returned
375      * as the value of this field.
376      * As such, day-of-year 1 to 7 will have aligned-day-of-week values from 1 to 7.
377      * And day-of-year 8 to 14 will repeat this with aligned-day-of-week values from 1 to 7.
378      * <p>
379      * Calendar systems that do not have a seven day week should typically implement this
380      * field in the same way, but using the alternate week length.
381      */
382     ALIGNED_DAY_OF_WEEK_IN_YEAR("AlignedDayOfWeekInYear", DAYS, WEEKS, ValueRange.of(1, 7)),
383     /**
384      * The day-of-month.
385      * <p>
386      * This represents the concept of the day within the month.
387      * In the default ISO calendar system, this has values from 1 to 31 in most months.
388      * April, June, September, November have days from 1 to 30, while February has days
389      * from 1 to 28, or 29 in a leap year.
390      * <p>
391      * Non-ISO calendar systems should implement this field using the most recognized
392      * day-of-month values for users of the calendar system.
393      * Normally, this is a count of days from 1 to the length of the month.
394      */
395     DAY_OF_MONTH("DayOfMonth", DAYS, MONTHS, ValueRange.of(1, 28, 31), "day"),
396     /**
397      * The day-of-year.

```

```

398 * <p>
399 * This represents the concept of the day within the year.
400 * In the default ISO calendar system, this has values from 1 to 365 in standard
401 * years and 1 to 366 in leap years.
402 * <p>
403 * Non-ISO calendar systems should implement this field using the most recognized
404 * day-of-year values for users of the calendar system.
405 * Normally, this is a count of days from 1 to the length of the year.
406 * <p>
407 * Note that a non-ISO calendar system may have year numbering system that changes
408 * at a different point to the natural reset in the month numbering. An example
409 * of this is the Japanese calendar system where a change of era, which resets
410 * the year number to 1, can happen on any date. The era and year reset also cause
411 * the day-of-year to be reset to 1, but not the month-of-year or day-of-month.
412 */
413 DAY_OF_YEAR("DayOfYear", DAYS, YEARS, ValueRange.of(1, 365, 366)),
414 /**
415 * The epoch-day, based on the Java epoch of 1970-01-01 (ISO).
416 * <p>
417 * This field is the sequential count of days where 1970-01-01 (ISO) is zero.
418 * Note that this uses the <i>local</i> time-line, ignoring offset and time-zone.
419 * <p>
420 * This field is strictly defined to have the same meaning in all calendar systems.
421 * This is necessary to ensure interoperation between calendars.
422 */
423 EPOCH_DAY("EpochDay", DAYS, FOREVER, ValueRange.of((long) (Year.MIN_VALUE * 365.25), (long) (
    Year.MAX_VALUE * 365.25))),
424 /**
425 * The aligned week within a month.
426 * <p>
427 * This represents concept of the count of weeks within the period of a month
428 * where the weeks are aligned to the start of the month.
429 * This field is typically used with {@link #ALIGNED_DAY_OF_WEEK_IN_MONTH}.
430 * <p>
431 * For example, in a calendar systems with a seven day week, the first aligned-week-of-month
432 * starts on day-of-month 1, the second aligned-week starts on day-of-month 8, and so on.
433 * Thus, day-of-month values 1 to 7 are in aligned-week 1, while day-of-month values
434 * 8 to 14 are in aligned-week 2, and so on.
435 * <p>
436 * Calendar systems that do not have a seven day week should typically implement this
437 * field in the same way, but using the alternate week length.
438 */
439 ALIGNED_WEEK_OF_MONTH("AlignedWeekOfMonth", WEEKS, MONTHS, ValueRange.of(1, 4, 5)),
440 /**
441 * The aligned week within a year.
442 * <p>
443 * This represents concept of the count of weeks within the period of a year
444 * where the weeks are aligned to the start of the year.
445 * This field is typically used with {@link #ALIGNED_DAY_OF_WEEK_IN_YEAR}.
446 * <p>
447 * For example, in a calendar systems with a seven day week, the first aligned-week-of-year
448 * starts on day-of-year 1, the second aligned-week starts on day-of-year 8, and so on.
449 * Thus, day-of-year values 1 to 7 are in aligned-week 1, while day-of-year values

```

```

450     * 8 to 14 are in aligned-week 2, and so on.
451     * <p>
452     * Calendar systems that do not have a seven day week should typically implement this
453     * field in the same way, but using the alternate week length.
454     */
455     ALIGNED_WEEK_OF_YEAR("AlignedWeekOfYear", WEEKS, YEARS, ValueRange.of(1, 53)),
456     /**
457     * The month-of-year, such as March.
458     * <p>
459     * This represents the concept of the month within the year.
460     * In the default ISO calendar system, this has values from January (1) to December (12).
461     * <p>
462     * Non-ISO calendar systems should implement this field using the most recognized
463     * month-of-year values for users of the calendar system.
464     * Normally, this is a count of months starting from 1.
465     */
466     MONTH_OF_YEAR("MonthOfYear", MONTHS, YEARS, ValueRange.of(1, 12), "month"),
467     /**
468     * The proleptic-month based, counting months sequentially from year 0.
469     * <p>
470     * This field is the sequential count of months where the first month
471     * in proleptic-year zero has the value zero.
472     * Later months have increasingly larger values.
473     * Earlier months have increasingly small values.
474     * There are no gaps or breaks in the sequence of months.
475     * Note that this uses the <i>local</i> time-line, ignoring offset and time-zone.
476     * <p>
477     * In the default ISO calendar system, June 2012 would have the value
478     * {@code (2012 * 12 + 6 - 1)}. This field is primarily for internal use.
479     * <p>
480     * Non-ISO calendar systems must implement this field as per the definition above.
481     * It is just a simple zero-based count of elapsed months from the start of proleptic-year 0.
482     * All calendar systems with a full proleptic-year definition will have a year zero.
483     * If the calendar system has a minimum year that excludes year zero, then one must
484     * be extrapolated in order for this method to be defined.
485     */
486     PROLEPTIC_MONTH("ProlepticMonth", MONTHS, FOREVER, ValueRange.of(Year.MIN_VALUE * 12L, Year.
487         MAX_VALUE * 12L + 11)),
488     /**
489     * The year within the era.
490     * <p>
491     * This represents the concept of the year within the era.
492     * This field is typically used with {@link #ERA}.
493     * <p>
494     * The standard mental model for a date is based on three concepts - year, month and day.
495     * These map onto the {@code YEAR}, {@code MONTH_OF_YEAR} and {@code DAY_OF_MONTH} fields.
496     * Note that there is no reference to eras.
497     * The full model for a date requires four concepts - era, year, month and day. These map onto
498     * the {@code ERA}, {@code YEAR_OF_ERA}, {@code MONTH_OF_YEAR} and {@code DAY_OF_MONTH} fields.
499     * Whether this field or {@code YEAR} is used depends on which mental model is being used.
500     * See {@link ChronoLocalDate} for more discussion on this topic.
501     * <p>
502     * In the default ISO calendar system, there are two eras defined, 'BCE' and 'CE'.

```

```

502 * The era 'CE' is the one currently in use and year-of-era runs from 1 to the maximum value.
503 * The era 'BCE' is the previous era, and the year-of-era runs backwards.
504 * <p>
505 * For example, subtracting a year each time yield the following:<br>
506 * - year-proleptic 2 = 'CE' year-of-era 2<br>
507 * - year-proleptic 1 = 'CE' year-of-era 1<br>
508 * - year-proleptic 0 = 'BCE' year-of-era 1<br>
509 * - year-proleptic -1 = 'BCE' year-of-era 2<br>
510 * <p>
511 * Note that the ISO-8601 standard does not actually define eras.
512 * Note also that the ISO eras do not align with the well-known AD/BC eras due to the
513 * change between the Julian and Gregorian calendar systems.
514 * <p>
515 * Non-ISO calendar systems should implement this field using the most recognized
516 * year-of-era value for users of the calendar system.
517 * Since most calendar systems have only two eras, the year-of-era numbering approach
518 * will typically be the same as that used by the ISO calendar system.
519 * The year-of-era value should typically always be positive, however this is not required.
520 */
521 YEAR_OF_ERA("YearOfEra", YEARS, FOREVER, ValueRange.of(1, Year.MAX_VALUE, Year.MAX_VALUE + 1)),
522 /**
523 * The proleptic year, such as 2012.
524 * <p>
525 * This represents the concept of the year, counting sequentially and using negative numbers.
526 * The proleptic year is not interpreted in terms of the era.
527 * See {@link #YEAR_OF_ERA} for an example showing the mapping from proleptic year to year-of-
    era.
528 * <p>
529 * The standard mental model for a date is based on three concepts - year, month and day.
530 * These map onto the {@code YEAR}, {@code MONTH_OF_YEAR} and {@code DAY_OF_MONTH} fields.
531 * Note that there is no reference to eras.
532 * The full model for a date requires four concepts - era, year, month and day. These map onto
533 * the {@code ERA}, {@code YEAR_OF_ERA}, {@code MONTH_OF_YEAR} and {@code DAY_OF_MONTH} fields.
534 * Whether this field or {@code YEAR_OF_ERA} is used depends on which mental model is being
    used.
535 * See {@link ChronoLocalDate} for more discussion on this topic.
536 * <p>
537 * Non-ISO calendar systems should implement this field as follows.
538 * If the calendar system has only two eras, before and after a fixed date, then the
539 * proleptic-year value must be the same as the year-of-era value for the later era,
540 * and increasingly negative for the earlier era.
541 * If the calendar system has more than two eras, then the proleptic-year value may be
542 * defined with any appropriate value, although defining it to be the same as ISO may be
543 * the best option.
544 */
545 YEAR("Year", YEARS, FOREVER, ValueRange.of(Year.MIN_VALUE, Year.MAX_VALUE), "year"),
546 /**
547 * The era.
548 * <p>
549 * This represents the concept of the era, which is the largest division of the time-line.
550 * This field is typically used with {@link #YEAR_OF_ERA}.
551 * <p>
552 * In the default ISO calendar system, there are two eras defined, 'BCE' and 'CE'.

```



```

553     * The era 'CE' is the one currently in use and year-of-era runs from 1 to the maximum value.
554     * The era 'BCE' is the previous era, and the year-of-era runs backwards.
555     * See {@link #YEAR_OF_ERA} for a full example.
556     * <p>
557     * Non-ISO calendar systems should implement this field to define eras.
558     * The value of the era that was active on 1970-01-01 (ISO) must be assigned the value 1.
559     * Earlier eras must have sequentially smaller values.
560     * Later eras must have sequentially larger values,
561     */
562     ERA("Era", ERAS, FOREVER, ValueRange.of(0, 1), "era"),
563     /**
564     * The instant epoch-seconds.
565     * <p>
566     * This represents the concept of the sequential count of seconds where
567     * 1970-01-01T00:00Z (ISO) is zero.
568     * This field may be used with {@link #NANO_OF_SECOND} to represent the fraction of the second.
569     * <p>
570     * An {@link Instant} represents an instantaneous point on the time-line.
571     * On their own, an instant has insufficient information to allow a local date-time to be
572     * obtained.
573     * Only when paired with an offset or time-zone can the local date or time be calculated.
574     * <p>
575     * This field is strictly defined to have the same meaning in all calendar systems.
576     * This is necessary to ensure interoperation between calendars.
577     */
578     INSTANT_SECONDS("InstantSeconds", SECONDS, FOREVER, ValueRange.of(Long.MIN_VALUE, Long.
579         MAX_VALUE)),
580     /**
581     * The offset from UTC/Greenwich.
582     * <p>
583     * This represents the concept of the offset in seconds of local time from UTC/Greenwich.
584     * <p>
585     * A {@link ZoneOffset} represents the period of time that local time differs from UTC/
586     * Greenwich.
587     * This is usually a fixed number of hours and minutes.
588     * It is equivalent to the {@link ZoneOffset#getTotalSeconds() total amount} of the offset in
589     * seconds.
590     * For example, during the winter Paris has an offset of {@code +01:00}, which is 3600 seconds.
591     * <p>
592     * This field is strictly defined to have the same meaning in all calendar systems.
593     * This is necessary to ensure interoperation between calendars.
594     */
595     OFFSET_SECONDS("OffsetSeconds", SECONDS, FOREVER, ValueRange.of(-18 * 3600, 18 * 3600));
596
597     private final String name;
598     private final TemporalUnit baseUnit;
599     private final TemporalUnit rangeUnit;
600     private final ValueRange range;
601     private final String displayNameKey;
602
603     private ChronoField(String name, TemporalUnit baseUnit, TemporalUnit rangeUnit, ValueRange
604         range) {
605         this.name = name;

```

```

601     this.baseUnit = baseUnit;
602     this.rangeUnit = rangeUnit;
603     this.range = range;
604     this.displayNameKey = null;
605 }
606
607 private ChronoField(String name, TemporalUnit baseUnit, TemporalUnit rangeUnit,
608     ValueRange range, String displayNameKey) {
609     this.name = name;
610     this.baseUnit = baseUnit;
611     this.rangeUnit = rangeUnit;
612     this.range = range;
613     this.displayNameKey = displayNameKey;
614 }
615
616 @Override
617 public String getDisplayName(Locale locale) {
618     Objects.requireNonNull(locale, "locale");
619     if (displayNameKey == null) {
620         return name;
621     }
622
623     LocaleResources lr = LocaleProviderAdapter.getResourceBundleBased()
624         .getLocaleResources(locale);
625     ResourceBundle rb = lr.getJavaTimeFormatData();
626     String key = "field." + displayNameKey;
627     return rb.containsKey(key) ? rb.getString(key) : name;
628 }
629
630 @Override
631 public TemporalUnit getBaseUnit() {
632     return baseUnit;
633 }
634
635 @Override
636 public TemporalUnit getRangeUnit() {
637     return rangeUnit;
638 }
639
640 /**
641  * Gets the range of valid values for the field.
642  * <p>
643  * All fields can be expressed as a {@code long} integer.
644  * This method returns an object that describes the valid range for that value.
645  * <p>
646  * This method returns the range of the field in the ISO-8601 calendar system.
647  * This range may be incorrect for other calendar systems.
648  * Use {@link Chronology#range(ChronoField)} to access the correct range
649  * for a different calendar system.
650  * <p>
651  * Note that the result only describes the minimum and maximum valid values
652  * and it is important not to read too much into them. For example, there
653  * could be values within the range that are invalid for the field.

```



```

654     *
655     * @return the range of valid values for the field, not null
656     */
657     @Override
658     public ValueRange range() {
659         return range;
660     }
661
662     //-----
663     /**
664     * Checks if this field represents a component of a date.
665     * <p>
666     * Fields from day-of-week to era are date-based.
667     *
668     * @return true if it is a component of a date
669     */
670     @Override
671     public boolean isDateBased() {
672         return ordinal() >= DAY_OF_WEEK.ordinal() && ordinal() <= ERA.ordinal();
673     }
674
675     /**
676     * Checks if this field represents a component of a time.
677     * <p>
678     * Fields from nano-of-second to am-pm-of-day are time-based.
679     *
680     * @return true if it is a component of a time
681     */
682     @Override
683     public boolean isTimeBased() {
684         return ordinal() < DAY_OF_WEEK.ordinal();
685     }
686
687     //-----
688     /**
689     * Checks that the specified value is valid for this field.
690     * <p>
691     * This validates that the value is within the outer range of valid values
692     * returned by {@link #range()}.
693     * <p>
694     * This method checks against the range of the field in the ISO-8601 calendar system.
695     * This range may be incorrect for other calendar systems.
696     * Use {@link Chronology#range(ChronoField)} to access the correct range
697     * for a different calendar system.
698     *
699     * @param value the value to check
700     * @return the value that was passed in
701     */
702     public long checkValidValue(long value) {
703         return range().checkValidValue(value, this);
704     }
705
706     /**

```

```

707     * Checks that the specified value is valid and fits in an {@code int}.
708     * <p>
709     * This validates that the value is within the outer range of valid values
710     * returned by {@link #range()}.
711     * It also checks that all valid values are within the bounds of an {@code int}.
712     * <p>
713     * This method checks against the range of the field in the ISO-8601 calendar system.
714     * This range may be incorrect for other calendar systems.
715     * Use {@link Chronology#range(ChronoField)} to access the correct range
716     * for a different calendar system.
717     *
718     * @param value the value to check
719     * @return the value that was passed in
720     */
721     public int checkValidIntValue(long value) {
722         return range().checkValidIntValue(value, this);
723     }
724
725     //-----
726     @Override
727     public boolean isSupportedBy(TemporalAccessor temporal) {
728         return temporal.isSupported(this);
729     }
730
731     @Override
732     public ValueRange rangeRefinedBy(TemporalAccessor temporal) {
733         return temporal.range(this);
734     }
735
736     @Override
737     public long getFrom(TemporalAccessor temporal) {
738         return temporal.getLong(this);
739     }
740
741     @SuppressWarnings("unchecked")
742     @Override
743     public <R extends Temporal> R adjustInto(R temporal, long newValue) {
744         return (R) temporal.with(this, newValue);
745     }
746
747     //-----
748     @Override
749     public String toString() {
750         return name;
751     }
752
753 }

```

7.2 ChronoUnit.java

Secuencia 113: java/time/temporal/ChronoUnit.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.

```

```
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27  * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
28  *
29  * All rights reserved.
30  *
31  * Redistribution and use in source and binary forms, with or without
32  * modification, are permitted provided that the following conditions are met:
33  *
34  * * Redistributions of source code must retain the above copyright notice,
35  *   this list of conditions and the following disclaimer.
36  *
37  * * Redistributions in binary form must reproduce the above copyright notice,
38  *   this list of conditions and the following disclaimer in the documentation
39  *   and/or other materials provided with the distribution.
40  *
41  * * Neither the name of JSR-310 nor the names of its contributors
42  *   may be used to endorse or promote products derived from this software
43  *   without specific prior written permission.
44  *
45  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

```
56  */
57  package java.time.temporal;
58
59  import java.time.Duration;
60
61  /**
62   * A standard set of date periods units.
63   * <p>
64   * This set of units provide unit-based access to manipulate a date, time or date-time.
65   * The standard set of units can be extended by implementing {@link TemporalUnit}.
66   * <p>
67   * These units are intended to be applicable in multiple calendar systems.
68   * For example, most non-ISO calendar systems define units of years, months and days,
69   * just with slightly different rules.
70   * The documentation of each unit explains how it operates.
71   *
72   * @implSpec
73   * This is a final, immutable and thread-safe enum.
74   *
75   * @since 1.8
76   */
77  public enum ChronoUnit implements TemporalUnit {
78
79      /**
80       * Unit that represents the concept of a nanosecond, the smallest supported unit of time.
81       * For the ISO calendar system, it is equal to the 1,000,000,000th part of the second unit.
82       */
83      NANOS("Nanos", Duration.ofNanos(1)),
84      /**
85       * Unit that represents the concept of a microsecond.
86       * For the ISO calendar system, it is equal to the 1,000,000th part of the second unit.
87       */
88      MICROS("Micros", Duration.ofNanos(1000)),
89      /**
90       * Unit that represents the concept of a millisecond.
91       * For the ISO calendar system, it is equal to the 1000th part of the second unit.
92       */
93      MILLIS("Millis", Duration.ofNanos(1000_000)),
94      /**
95       * Unit that represents the concept of a second.
96       * For the ISO calendar system, it is equal to the second in the SI system
97       * of units, except around a leap-second.
98       */
99      SECONDS("Seconds", Duration.ofSeconds(1)),
100     /**
101      * Unit that represents the concept of a minute.
102      * For the ISO calendar system, it is equal to 60 seconds.
103      */
104     MINUTES("Minutes", Duration.ofSeconds(60)),
105     /**
106      * Unit that represents the concept of an hour.
107      * For the ISO calendar system, it is equal to 60 minutes.
108      */
109 }
```

```

109 HOURS("Hours", Duration.ofSeconds(3600)),
110 /**
111  * Unit that represents the concept of half a day, as used in AM/PM.
112  * For the ISO calendar system, it is equal to 12 hours.
113  */
114 HALF_DAYS("HalfDays", Duration.ofSeconds(43200)),
115 /**
116  * Unit that represents the concept of a day.
117  * For the ISO calendar system, it is the standard day from midnight to midnight.
118  * The estimated duration of a day is {@code 24 Hours}.
119  * <p>
120  * When used with other calendar systems it must correspond to the day defined by
121  * the rising and setting of the Sun on Earth. It is not required that days begin
122  * at midnight - when converting between calendar systems, the date should be
123  * equivalent at midday.
124  */
125 DAYS("Days", Duration.ofSeconds(86400)),
126 /**
127  * Unit that represents the concept of a week.
128  * For the ISO calendar system, it is equal to 7 days.
129  * <p>
130  * When used with other calendar systems it must correspond to an integral number of days.
131  */
132 WEEKS("Weeks", Duration.ofSeconds(7 * 86400L)),
133 /**
134  * Unit that represents the concept of a month.
135  * For the ISO calendar system, the length of the month varies by month-of-year.
136  * The estimated duration of a month is one twelfth of {@code 365.2425 Days}.
137  * <p>
138  * When used with other calendar systems it must correspond to an integral number of days.
139  */
140 MONTHS("Months", Duration.ofSeconds(31556952L / 12)),
141 /**
142  * Unit that represents the concept of a year.
143  * For the ISO calendar system, it is equal to 12 months.
144  * The estimated duration of a year is {@code 365.2425 Days}.
145  * <p>
146  * When used with other calendar systems it must correspond to an integral number of days
147  * or months roughly equal to a year defined by the passage of the Earth around the Sun.
148  */
149 YEARS("Years", Duration.ofSeconds(31556952L)),
150 /**
151  * Unit that represents the concept of a decade.
152  * For the ISO calendar system, it is equal to 10 years.
153  * <p>
154  * When used with other calendar systems it must correspond to an integral number of days
155  * and is normally an integral number of years.
156  */
157 DECADES("Decades", Duration.ofSeconds(31556952L * 10L)),
158 /**
159  * Unit that represents the concept of a century.
160  * For the ISO calendar system, it is equal to 100 years.
161  * <p>

```

```

162     * When used with other calendar systems it must correspond to an integral number of days
163     * and is normally an integral number of years.
164     */
165     CENTURIES("Centuries", Duration.ofSeconds(31556952L * 100L)),
166     /**
167     * Unit that represents the concept of a millennium.
168     * For the ISO calendar system, it is equal to 1000 years.
169     * <p>
170     * When used with other calendar systems it must correspond to an integral number of days
171     * and is normally an integral number of years.
172     */
173     MILLENNIA("Millennia", Duration.ofSeconds(31556952L * 1000L)),
174     /**
175     * Unit that represents the concept of an era.
176     * The ISO calendar system doesn't have eras thus it is impossible to add
177     * an era to a date or date-time.
178     * The estimated duration of the era is artificially defined as {@code 1,000,000,000 Years}.
179     * <p>
180     * When used with other calendar systems there are no restrictions on the unit.
181     */
182     ERAS("Eras", Duration.ofSeconds(31556952L * 1000_000_000L)),
183     /**
184     * Artificial unit that represents the concept of forever.
185     * This is primarily used with {@link TemporalField} to represent unbounded fields
186     * such as the year or era.
187     * The estimated duration of the era is artificially defined as the largest duration
188     * supported by {@code Duration}.
189     */
190     FOREVER("Forever", Duration.ofSeconds(Long.MAX_VALUE, 999_999_999));
191
192     private final String name;
193     private final Duration duration;
194
195     private ChronoUnit(String name, Duration estimatedDuration) {
196         this.name = name;
197         this.duration = estimatedDuration;
198     }
199
200     //-----
201     /**
202     * Gets the estimated duration of this unit in the ISO calendar system.
203     * <p>
204     * All of the units in this class have an estimated duration.
205     * Days vary due to daylight saving time, while months have different lengths.
206     *
207     * @return the estimated duration of this unit, not null
208     */
209     @Override
210     public Duration getDuration() {
211         return duration;
212     }
213
214     /**

```

```

215     * Checks if the duration of the unit is an estimate.
216     * <p>
217     * All time units in this class are considered to be accurate, while all date
218     * units in this class are considered to be estimated.
219     * <p>
220     * This definition ignores leap seconds, but considers that Days vary due to
221     * daylight saving time and months have different lengths.
222     *
223     * @return true if the duration is estimated, false if accurate
224     */
225     @Override
226     public boolean isDurationEstimated() {
227         return this.compareTo(DAYS) >= 0;
228     }
229
230     //-----
231     /**
232     * Checks if this unit is a date unit.
233     * <p>
234     * All units from days to eras inclusive are date-based.
235     * Time-based units and {@code FOREVER} return false.
236     *
237     * @return true if a date unit, false if a time unit
238     */
239     @Override
240     public boolean isDateBased() {
241         return this.compareTo(DAYS) >= 0 && this != FOREVER;
242     }
243
244     /**
245     * Checks if this unit is a time unit.
246     * <p>
247     * All units from nanos to half-days inclusive are time-based.
248     * Date-based units and {@code FOREVER} return false.
249     *
250     * @return true if a time unit, false if a date unit
251     */
252     @Override
253     public boolean isTimeBased() {
254         return this.compareTo(DAYS) < 0;
255     }
256
257     //-----
258     @Override
259     public boolean isSupportedBy(Temporal temporal) {
260         return temporal.isSupported(this);
261     }
262
263     @SuppressWarnings("unchecked")
264     @Override
265     public <R extends Temporal> R addTo(R temporal, long amount) {
266         return (R) temporal.plus(amount, this);
267     }

```

```

268
269 //-----
270 @Override
271 public long between(Temporal temporal1Inclusive, Temporal temporal2Exclusive) {
272     return temporal1Inclusive.until(temporal2Exclusive, this);
273 }
274
275 //-----
276 @Override
277 public String toString() {
278     return name;
279 }
280
281 }

```

7.3 IsoFields.java

Secuencia 114: java/time/temporal/IsoFields.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 * Copyright (c) 2011-2012, Stephen Colebourne & Michael Nascimento Santos
28 *
29 * All rights reserved.
30 *
31 * Redistribution and use in source and binary forms, with or without
32 * modification, are permitted provided that the following conditions are met:
33 *
34 * * Redistributions of source code must retain the above copyright notice,
35 *   this list of conditions and the following disclaimer.

```



```
36  *
37  * * Redistributions in binary form must reproduce the above copyright notice,
38  *   this list of conditions and the following disclaimer in the documentation
39  *   and/or other materials provided with the distribution.
40  *
41  * * Neither the name of JSR-310 nor the names of its contributors
42  *   may be used to endorse or promote products derived from this software
43  *   without specific prior written permission.
44  *
45  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56  */
57 package java.time.temporal;
58
59 import static java.time.DayOfWeek.THURSDAY;
60 import static java.time.DayOfWeek.WEDNESDAY;
61 import static java.time.temporal.ChronoField.DAY_OF_WEEK;
62 import static java.time.temporal.ChronoField.DAY_OF_YEAR;
63 import static java.time.temporal.ChronoField.EPOCH_DAY;
64 import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
65 import static java.time.temporal.ChronoField.YEAR;
66 import static java.time.temporal.ChronoUnit.DAYS;
67 import static java.time.temporal.ChronoUnit.FOREVER;
68 import static java.time.temporal.ChronoUnit.MONTHS;
69 import static java.time.temporal.ChronoUnit.WEEKS;
70 import static java.time.temporal.ChronoUnit.YEARS;
71
72 import java.time.DateTimeException;
73 import java.time.Duration;
74 import java.time.LocalDate;
75 import java.time.chrono.ChronoLocalDate;
76 import java.time.chrono.Chronology;
77 import java.time.chrono.IsoChronology;
78 import java.time.format.ResolverStyle;
79 import java.util.Locale;
80 import java.util.Map;
81 import java.util.Objects;
82 import java.util.ResourceBundle;
83
84 import sun.util.locale.provider.LocaleProviderAdapter;
85 import sun.util.locale.provider.LocaleResources;
86
87 /**
88  * Fields and units specific to the ISO-8601 calendar system,
```

```

89 * including quarter-of-year and week-based-year.
90 * <p>
91 * This class defines fields and units that are specific to the ISO calendar system.
92 *
93 * <h3>Quarter of year</h3>
94 * The ISO-8601 standard is based on the standard civic 12 month year.
95 * This is commonly divided into four quarters, often abbreviated as Q1, Q2, Q3 and Q4.
96 * <p>
97 * January, February and March are in Q1.
98 * April, May and June are in Q2.
99 * July, August and September are in Q3.
100 * October, November and December are in Q4.
101 * <p>
102 * The complete date is expressed using three fields:
103 * <ul>
104 * <li>{@link #DAY_OF_QUARTER DAY_OF_QUARTER} - the day within the quarter, from 1 to 90, 91 or 92
105 * <li>{@link #QUARTER_OF_YEAR QUARTER_OF_YEAR} - the week within the week-based-year
106 * <li>{@link ChronoField#YEAR YEAR} - the standard ISO year
107 * </ul>
108 *
109 * <h3>Week based years</h3>
110 * The ISO-8601 standard was originally intended as a data interchange format,
111 * defining a string format for dates and times. However, it also defines an
112 * alternate way of expressing the date, based on the concept of week-based-year.
113 * <p>
114 * The date is expressed using three fields:
115 * <ul>
116 * <li>{@link ChronoField#DAY_OF_WEEK DAY_OF_WEEK} - the standard field defining the
117 * day-of-week from Monday (1) to Sunday (7)
118 * <li>{@link #WEEK_OF_WEEK_BASED_YEAR} - the week within the week-based-year
119 * <li>{@link #WEEK_BASED_YEAR WEEK_BASED_YEAR} - the week-based-year
120 * </ul>
121 * The week-based-year itself is defined relative to the standard ISO proleptic year.
122 * It differs from the standard year in that it always starts on a Monday.
123 * <p>
124 * The first week of a week-based-year is the first Monday-based week of the standard
125 * ISO year that has at least 4 days in the new year.
126 * <ul>
127 * <li>If January 1st is Monday then week 1 starts on January 1st
128 * <li>If January 1st is Tuesday then week 1 starts on December 31st of the previous standard year
129 * <li>If January 1st is Wednesday then week 1 starts on December 30th of the previous standard
    year
130 * <li>If January 1st is Thursday then week 1 starts on December 29th of the previous standard year
131 * <li>If January 1st is Friday then week 1 starts on January 4th
132 * <li>If January 1st is Saturday then week 1 starts on January 3rd
133 * <li>If January 1st is Sunday then week 1 starts on January 2nd
134 * </ul>
135 * There are 52 weeks in most week-based years, however on occasion there are 53 weeks.
136 * <p>
137 * For example:
138 *
139 * <table cellpadding="0" cellspacing="3" border="0" style="text-align: left; width: 50%;">
140 * <caption>Examples of Week based Years</caption>

```

```

141 * <tr><th>Date</th><th>Day-of-week</th><th>Field values</th></tr>
142 * <tr><th>2008-12-28</th><td>Sunday</td><td>Week 52 of week-based-year 2008</td></tr>
143 * <tr><th>2008-12-29</th><td>Monday</td><td>Week 1 of week-based-year 2009</td></tr>
144 * <tr><th>2008-12-31</th><td>Wednesday</td><td>Week 1 of week-based-year 2009</td></tr>
145 * <tr><th>2009-01-01</th><td>Thursday</td><td>Week 1 of week-based-year 2009</td></tr>
146 * <tr><th>2009-01-04</th><td>Sunday</td><td>Week 1 of week-based-year 2009</td></tr>
147 * <tr><th>2009-01-05</th><td>Monday</td><td>Week 2 of week-based-year 2009</td></tr>
148 * </table>
149 *
150 * @implSpec
151 * <p>
152 * This class is immutable and thread-safe.
153 *
154 * @since 1.8
155 */
156 public final class IsoFields {
157
158     /**
159      * The field that represents the day-of-quarter.
160      * <p>
161      * This field allows the day-of-quarter value to be queried and set.
162      * The day-of-quarter has values from 1 to 90 in Q1 of a standard year, from 1 to 91
163      * in Q1 of a leap year, from 1 to 91 in Q2 and from 1 to 92 in Q3 and Q4.
164      * <p>
165      * The day-of-quarter can only be calculated if the day-of-year, month-of-year and year
166      * are available.
167      * <p>
168      * When setting this field, the value is allowed to be partially lenient, taking any
169      * value from 1 to 92. If the quarter has less than 92 days, then day 92, and
170      * potentially day 91, is in the following quarter.
171      * <p>
172      * In the resolving phase of parsing, a date can be created from a year,
173      * quarter-of-year and day-of-quarter.
174      * <p>
175      * In {@linkplain ResolverStyle#STRICT strict mode}, all three fields are
176      * validated against their range of valid values. The day-of-quarter field
177      * is validated from 1 to 90, 91 or 92 depending on the year and quarter.
178      * <p>
179      * In {@linkplain ResolverStyle#SMART smart mode}, all three fields are
180      * validated against their range of valid values. The day-of-quarter field is
181      * validated between 1 and 92, ignoring the actual range based on the year and quarter.
182      * If the day-of-quarter exceeds the actual range by one day, then the resulting date
183      * is one day later. If the day-of-quarter exceeds the actual range by two days,
184      * then the resulting date is two days later.
185      * <p>
186      * In {@linkplain ResolverStyle#LENIENT lenient mode}, only the year is validated
187      * against the range of valid values. The resulting date is calculated equivalent to
188      * the following three stage approach. First, create a date on the first of January
189      * in the requested year. Then take the quarter-of-year, subtract one, and add the
190      * amount in quarters to the date. Finally, take the day-of-quarter, subtract one,
191      * and add the amount in days to the date.
192      * <p>
193      * This unit is an immutable and thread-safe singleton.

```

```

194     */
195     public static final TemporalField DAY_OF_QUARTER = Field.DAY_OF_QUARTER;
196     /**
197      * The field that represents the quarter-of-year.
198      * <p>
199      * This field allows the quarter-of-year value to be queried and set.
200      * The quarter-of-year has values from 1 to 4.
201      * <p>
202      * The quarter-of-year can only be calculated if the month-of-year is available.
203      * <p>
204      * In the resolving phase of parsing, a date can be created from a year,
205      * quarter-of-year and day-of-quarter.
206      * See {@link #DAY_OF_QUARTER} for details.
207      * <p>
208      * This unit is an immutable and thread-safe singleton.
209      */
210     public static final TemporalField QUARTER_OF_YEAR = Field.QUARTER_OF_YEAR;
211     /**
212      * The field that represents the week-of-week-based-year.
213      * <p>
214      * This field allows the week of the week-based-year value to be queried and set.
215      * The week-of-week-based-year has values from 1 to 52, or 53 if the
216      * week-based-year has 53 weeks.
217      * <p>
218      * In the resolving phase of parsing, a date can be created from a
219      * week-based-year, week-of-week-based-year and day-of-week.
220      * <p>
221      * In {@linkplain ResolverStyle#STRICT strict mode}, all three fields are
222      * validated against their range of valid values. The week-of-week-based-year
223      * field is validated from 1 to 52 or 53 depending on the week-based-year.
224      * <p>
225      * In {@linkplain ResolverStyle#SMART smart mode}, all three fields are
226      * validated against their range of valid values. The week-of-week-based-year
227      * field is validated between 1 and 53, ignoring the week-based-year.
228      * If the week-of-week-based-year is 53, but the week-based-year only has
229      * 52 weeks, then the resulting date is in week 1 of the following week-based-year.
230      * <p>
231      * In {@linkplain ResolverStyle#LENIENT lenient mode}, only the week-based-year
232      * is validated against the range of valid values. If the day-of-week is outside
233      * the range 1 to 7, then the resulting date is adjusted by a suitable number of
234      * weeks to reduce the day-of-week to the range 1 to 7. If the week-of-week-based-year
235      * value is outside the range 1 to 52, then any excess weeks are added or subtracted
236      * from the resulting date.
237      * <p>
238      * This unit is an immutable and thread-safe singleton.
239      */
240     public static final TemporalField WEEK_OF_WEEK_BASED_YEAR = Field.WEEK_OF_WEEK_BASED_YEAR;
241     /**
242      * The field that represents the week-based-year.
243      * <p>
244      * This field allows the week-based-year value to be queried and set.
245      * <p>
246      * The field has a range that matches {@link LocalDate#MAX} and {@link LocalDate#MIN}.

```

```

247     * <p>
248     * In the resolving phase of parsing, a date can be created from a
249     * week-based-year, week-of-week-based-year and day-of-week.
250     * See {@link #WEEK_OF_WEEK_BASED_YEAR} for details.
251     * <p>
252     * This unit is an immutable and thread-safe singleton.
253     */
254     public static final TemporalField WEEK_BASED_YEAR = Field.WEEK_BASED_YEAR;
255     /**
256     * The unit that represents week-based-years for the purpose of addition and subtraction.
257     * <p>
258     * This allows a number of week-based-years to be added to, or subtracted from, a date.
259     * The unit is equal to either 52 or 53 weeks.
260     * The estimated duration of a week-based-year is the same as that of a standard ISO
261     * year at {@code 365.2425 Days}.
262     * <p>
263     * The rules for addition add the number of week-based-years to the existing value
264     * for the week-based-year field. If the resulting week-based-year only has 52 weeks,
265     * then the date will be in week 1 of the following week-based-year.
266     * <p>
267     * This unit is an immutable and thread-safe singleton.
268     */
269     public static final TemporalUnit WEEK_BASED_YEARS = Unit.WEEK_BASED_YEARS;
270     /**
271     * Unit that represents the concept of a quarter-year.
272     * For the ISO calendar system, it is equal to 3 months.
273     * The estimated duration of a quarter-year is one quarter of {@code 365.2425 Days}.
274     * <p>
275     * This unit is an immutable and thread-safe singleton.
276     */
277     public static final TemporalUnit QUARTER_YEARS = Unit.QUARTER_YEARS;
278
279     /**
280     * Restricted constructor.
281     */
282     private IsoFields() {
283         throw new AssertionError("Not instantiable");
284     }
285
286     //-----
287     /**
288     * Implementation of the field.
289     */
290     private static enum Field implements TemporalField {
291         DAY_OF_QUARTER {
292             @Override
293             public TemporalUnit getBaseUnit() {
294                 return DAYS;
295             }
296             @Override
297             public TemporalUnit getRangeUnit() {
298                 return QUARTER_YEARS;
299             }

```

```

300     @Override
301     public ValueRange range() {
302         return ValueRange.of(1, 90, 92);
303     }
304     @Override
305     public boolean isSupportedBy(TemporalAccessor temporal) {
306         return temporal.isSupported(DAY_OF_YEAR) && temporal.isSupported(MONTH_OF_YEAR) &&
307             temporal.isSupported(YEAR) && isIso(temporal);
308     }
309     @Override
310     public ValueRange rangeRefinedBy(TemporalAccessor temporal) {
311         if (isSupportedBy(temporal) == false) {
312             throw new UnsupportedTemporalTypeException("Unsupported field: DayOfQuarter");
313         }
314         long qoy = temporal.getLong(QUARTER_OF_YEAR);
315         if (qoy == 1) {
316             long year = temporal.getLong(YEAR);
317             return (IsoChronology.INSTANCE.isLeapYear(year) ? ValueRange.of(1, 91) :
318                 ValueRange.of(1, 90));
319         } else if (qoy == 2) {
320             return ValueRange.of(1, 91);
321         } else if (qoy == 3 || qoy == 4) {
322             return ValueRange.of(1, 92);
323         } // else value not from 1 to 4, so drop through
324         return range();
325     }
326     @Override
327     public long getFrom(TemporalAccessor temporal) {
328         if (isSupportedBy(temporal) == false) {
329             throw new UnsupportedTemporalTypeException("Unsupported field: DayOfQuarter");
330         }
331         int doy = temporal.get(DAY_OF_YEAR);
332         int moy = temporal.get(MONTH_OF_YEAR);
333         long year = temporal.getLong(YEAR);
334         return doy - QUARTER_DAYS[((moy - 1) / 3) + (IsoChronology.INSTANCE.isLeapYear(year)
335             ? 4 : 0)];
336     }
337     @SuppressWarnings("unchecked")
338     @Override
339     public <R extends Temporal> R adjustInto(R temporal, long newValue) {
340         // calls getFrom() to check if supported
341         long curValue = getFrom(temporal);
342         range().checkValidValue(newValue, this); // leniently check from 1 to 92 TODO:
343             check
344         return (R) temporal.with(DAY_OF_YEAR, temporal.getLong(DAY_OF_YEAR) + (newValue -
345             curValue));
346     }
347     @Override
348     public ChronoLocalDate resolve(
349         Map<TemporalField, Long> fieldValues, TemporalAccessor partialTemporal,
350         ResolverStyle resolverStyle) {
351         Long yearLong = fieldValues.get(YEAR);
352         Long qoyLong = fieldValues.get(QUARTER_OF_YEAR);

```

```

348         if (yearLong == null || qoyLong == null) {
349             return null;
350         }
351         int y = YEAR.checkValidIntValue(yearLong); // always validate
352         long doq = fieldValues.get(DAY_OF_QUARTER);
353         ensureIso(partialTemporal);
354         LocalDate date;
355         if (resolverStyle == ResolverStyle.LENIENT) {
356             date = LocalDate.of(y, 1, 1).plusMonths(Math.multiplyExact(Math.subtractExact(
357                 qoyLong, 1), 3));
358             doq = Math.subtractExact(doq, 1);
359         } else {
360             int qoy = QUARTER_OF_YEAR.range().checkValidIntValue(qoyLong, QUARTER_OF_YEAR);
361             // validated
362             date = LocalDate.of(y, ((qoy - 1) * 3) + 1, 1);
363             if (doq < 1 || doq > 90) {
364                 if (resolverStyle == ResolverStyle.STRICT) {
365                     rangeRefinedBy(date).checkValidValue(doq, this); // only allow exact
366                                     range
367                 } else { // SMART
368                     range().checkValidValue(doq, this); // allow 1-92 rolling into next
369                                     quarter
370                 }
371             }
372             doq--;
373         }
374         fieldValues.remove(this);
375         fieldValues.remove(YEAR);
376         fieldValues.remove(QUARTER_OF_YEAR);
377         return date.plusDays(doq);
378     }
379 },
380 QUARTER_OF_YEAR {
381     @Override
382     public TemporalUnit getBaseUnit() {
383         return QUARTER_YEARS;
384     }
385     @Override
386     public TemporalUnit getRangeUnit() {
387         return YEARS;
388     }
389     @Override
390     public ValueRange range() {
391         return ValueRange.of(1, 4);
392     }
393     @Override
394     public boolean isSupportedBy(TemporalAccessor temporal) {
395         return temporal.isSupported(MONTH_OF_YEAR) && isIso(temporal);
396     }

```



```

397     @Override
398     public long getFrom(TemporalAccessor temporal) {
399         if (isSupportedBy(temporal) == false) {
400             throw new UnsupportedOperationException("Unsupported field: QuarterOfYear");
401         }
402         long moy = temporal.getLong(MONTH_OF_YEAR);
403         return ((moy + 2) / 3);
404     }
405     @SuppressWarnings("unchecked")
406     @Override
407     public <R extends Temporal> R adjustInto(R temporal, long newValue) {
408         // calls getFrom() to check if supported
409         long curValue = getFrom(temporal);
410         range().checkValidValue(newValue, this); // strictly check from 1 to 4
411         return (R) temporal.with(MONTH_OF_YEAR, temporal.getLong(MONTH_OF_YEAR) + (newValue
412             - curValue) * 3);
413     }
414     @Override
415     public String toString() {
416         return "QuarterOfYear";
417     },
418     WEEK_OF_WEEK_BASED_YEAR {
419         @Override
420         public String getDisplayName(Locale locale) {
421             Objects.requireNonNull(locale, "locale");
422             LocaleResources lr = LocaleProviderAdapter.getResourceBundleBased()
423                 .getLocaleResources(locale);
424             ResourceBundle rb = lr.getJavaTimeFormatData();
425             return rb.containsKey("field.week") ? rb.getString("field.week") : toString();
426         }
427
428         @Override
429         public TemporalUnit getBaseUnit() {
430             return WEEKS;
431         }
432         @Override
433         public TemporalUnit getRangeUnit() {
434             return WEEK_BASED_YEARS;
435         }
436         @Override
437         public ValueRange range() {
438             return ValueRange.of(1, 52, 53);
439         }
440         @Override
441         public boolean isSupportedBy(TemporalAccessor temporal) {
442             return temporal.isSupported(EPOCH_DAY) && isIso(temporal);
443         }
444         @Override
445         public ValueRange rangeRefinedBy(TemporalAccessor temporal) {
446             if (isSupportedBy(temporal) == false) {
447                 throw new UnsupportedOperationException("Unsupported field:
448                     WeekOfWeekBasedYear");

```



```

448     }
449     return getWeekRange(LocalDate.from(temporal));
450 }
451 @Override
452 public long getFrom(TemporalAccessor temporal) {
453     if (isSupportedBy(temporal) == false) {
454         throw new UnsupportedTemporalTypeException("Unsupported field:
455             WeekOfWeekBasedYear");
456     }
457     return getWeek(LocalDate.from(temporal));
458 }
459 @SuppressWarnings("unchecked")
460 @Override
461 public <R extends Temporal> R adjustInto(R temporal, long newValue) {
462     // calls getFrom() to check if supported
463     range().checkValidValue(newValue, this); // lenient range
464     return (R) temporal.plus(Math.subtractExact(newValue, getFrom(temporal)), WEEKS);
465 }
466 @Override
467 public ChronoLocalDate resolve(
468     Map<TemporalField, Long> fieldValues, TemporalAccessor partialTemporal,
469     ResolverStyle resolverStyle) {
470     Long wbyLong = fieldValues.get(WEEK_BASED_YEAR);
471     Long dowLong = fieldValues.get(DAY_OF_WEEK);
472     if (wbyLong == null || dowLong == null) {
473         return null;
474     }
475     int wby = WEEK_BASED_YEAR.range().checkValidIntValue(wbyLong, WEEK_BASED_YEAR); //
476         always validate
477     long wowby = fieldValues.get(WEEK_OF_WEEK_BASED_YEAR);
478     ensureIso(partialTemporal);
479     LocalDate date = LocalDate.of(wby, 1, 4);
480     if (resolverStyle == ResolverStyle.LENIENT) {
481         long dow = dowLong; // unvalidated
482         if (dow > 7) {
483             date = date.plusWeeks((dow - 1) / 7);
484             dow = ((dow - 1) % 7) + 1;
485         } else if (dow < 1) {
486             date = date.plusWeeks(Math.subtractExact(dow, 7) / 7);
487             dow = ((dow + 6) % 7) + 1;
488         }
489         date = date.plusWeeks(Math.subtractExact(wowby, 1)).with(DAY_OF_WEEK, dow);
490     } else {
491         int dow = DAY_OF_WEEK.checkValidIntValue(dowLong); // validated
492         if (wowby < 1 || wowby > 52) {
493             if (resolverStyle == ResolverStyle.STRICT) {
494                 getWeekRange(date).checkValidValue(wowby, this); // only allow exact
495                 range
496             } else { // SMART
497                 range().checkValidValue(wowby, this); // allow 1-53 rolling into next
498                 year
499             }
500         }
501     }
502 }

```

```

496         date = date.plusWeeks(wowby - 1).with(DAY_OF_WEEK, dow);
497     }
498     fieldValues.remove(this);
499     fieldValues.remove(WEEK_BASED_YEAR);
500     fieldValues.remove(DAY_OF_WEEK);
501     return date;
502 }
503 @Override
504 public String toString() {
505     return "WeekOfWeekBasedYear";
506 }
507 },
508 WEEK_BASED_YEAR {
509     @Override
510     public TemporalUnit getBaseUnit() {
511         return WEEK_BASED_YEARS;
512     }
513     @Override
514     public TemporalUnit getRangeUnit() {
515         return FOREVER;
516     }
517     @Override
518     public ValueRange range() {
519         return YEAR.range();
520     }
521     @Override
522     public boolean isSupportedBy(TemporalAccessor temporal) {
523         return temporal.isSupported(EPOCH_DAY) && isIso(temporal);
524     }
525     @Override
526     public long getFrom(TemporalAccessor temporal) {
527         if (isSupportedBy(temporal) == false) {
528             throw new UnsupportedTemporalTypeException("Unsupported field: WeekBasedYear");
529         }
530         return getWeekBasedYear(LocalDate.from(temporal));
531     }
532     @SuppressWarnings("unchecked")
533     @Override
534     public <R extends Temporal> R adjustInto(R temporal, long newValue) {
535         if (isSupportedBy(temporal) == false) {
536             throw new UnsupportedTemporalTypeException("Unsupported field: WeekBasedYear");
537         }
538         int newWby = range().checkValidIntValue(newValue, WEEK_BASED_YEAR); // strict
539                                     check
540         LocalDate date = LocalDate.from(temporal);
541         int dow = date.get(DAY_OF_WEEK);
542         int week = getWeek(date);
543         if (week == 53 && getWeekRange(newWby) == 52) {
544             week = 52;
545         }
546         LocalDate resolved = LocalDate.of(newWby, 1, 4); // 4th is guaranteed to be in
547                                     week one
548         int days = (dow - resolved.get(DAY_OF_WEEK)) + ((week - 1) * 7);

```

```

547         resolved = resolved.plusDays(days);
548         return (R) temporal.with(resolved);
549     }
550     @Override
551     public String toString() {
552         return "WeekBasedYear";
553     }
554 };
555
556     @Override
557     public boolean isDateBased() {
558         return true;
559     }
560
561     @Override
562     public boolean isTimeBased() {
563         return false;
564     }
565
566     @Override
567     public ValueRange rangeRefinedBy(TemporalAccessor temporal) {
568         return range();
569     }
570
571     //-----
572     private static final int[] QUARTER_DAYS = {0, 90, 181, 273, 0, 91, 182, 274};
573
574     private static boolean isIso(TemporalAccessor temporal) {
575         return Chronology.from(temporal).equals(IsoChronology.INSTANCE);
576     }
577
578     private static void ensureIso(TemporalAccessor temporal) {
579         if (isIso(temporal) == false) {
580             throw new DateTimeException("Resolve requires IsoChronology");
581         }
582     }
583
584     private static ValueRange getWeekRange(LocalDate date) {
585         int wby = getWeekBasedYear(date);
586         return ValueRange.of(1, getWeekRange(wby));
587     }
588
589     private static int getWeekRange(int wby) {
590         LocalDate date = LocalDate.of(wby, 1, 1);
591         // 53 weeks if standard year starts on Thursday, or Wed in a leap year
592         if (date.getDayOfWeek() == THURSDAY || (date.getDayOfWeek() == WEDNESDAY && date.
593             isLeapYear())) {
594             return 53;
595         }
596         return 52;
597     }
598     private static int getWeek(LocalDate date) {

```

```

599     int dow0 = date.getDayOfWeek().ordinal();
600     int doy0 = date.getDayOfYear() - 1;
601     int doyThu0 = doy0 + (3 - dow0); // adjust to mid-week Thursday (which is 3 indexed
        from zero)
602     int alignedWeek = doyThu0 / 7;
603     int firstThuDoy0 = doyThu0 - (alignedWeek * 7);
604     int firstMonDoy0 = firstThuDoy0 - 3;
605     if (firstMonDoy0 < -3) {
606         firstMonDoy0 += 7;
607     }
608     if (doy0 < firstMonDoy0) {
609         return (int) getWeekRange(date.withDayOfYear(180).minusYears(1)).getMaximum();
610     }
611     int week = ((doy0 - firstMonDoy0) / 7) + 1;
612     if (week == 53) {
613         if ((firstMonDoy0 == -3 || (firstMonDoy0 == -2 && date.isLeapYear())) == false) {
614             week = 1;
615         }
616     }
617     return week;
618 }
619
620 private static int getWeekBasedYear(LocalDate date) {
621     int year = date.getYear();
622     int doy = date.getDayOfYear();
623     if (doy <= 3) {
624         int dow = date.getDayOfWeek().ordinal();
625         if (doy - dow < -2) {
626             year--;
627         }
628     } else if (doy >= 363) {
629         int dow = date.getDayOfWeek().ordinal();
630         doy = doy - 363 - (date.isLeapYear() ? 1 : 0);
631         if (doy - dow >= 0) {
632             year++;
633         }
634     }
635     return year;
636 }
637
638
639 //-----
640 /**
641  * Implementation of the unit.
642  */
643 private static enum Unit implements TemporalUnit {
644
645     /**
646      * Unit that represents the concept of a week-based-year.
647      */
648     WEEK_BASED_YEARS("WeekBasedYears", Duration.ofSeconds(31556952L)),
649     /**
650      * Unit that represents the concept of a quarter-year.

```

```

651     */
652     QUARTER_YEARS("QuarterYears", Duration.ofSeconds(31556952L / 4));
653
654     private final String name;
655     private final Duration duration;
656
657     private Unit(String name, Duration estimatedDuration) {
658         this.name = name;
659         this.duration = estimatedDuration;
660     }
661
662     @Override
663     public Duration getDuration() {
664         return duration;
665     }
666
667     @Override
668     public boolean isDurationEstimated() {
669         return true;
670     }
671
672     @Override
673     public boolean isDateBased() {
674         return true;
675     }
676
677     @Override
678     public boolean isTimeBased() {
679         return false;
680     }
681
682     @Override
683     public boolean isSupportedBy(Temporal temporal) {
684         return temporal.isSupported(EPOCH_DAY);
685     }
686
687     @SuppressWarnings("unchecked")
688     @Override
689     public <R extends Temporal> R addTo(R temporal, long amount) {
690         switch (this) {
691             case WEEK_BASED_YEARS:
692                 return (R) temporal.with(WEEK_BASED_YEAR,
693                     Math.addExact(temporal.get(WEEK_BASED_YEAR), amount));
694             case QUARTER_YEARS:
695                 // no overflow (256 is multiple of 4)
696                 return (R) temporal.plus(amount / 256, YEARS)
697                     .plus((amount % 256) * 3, MONTHS);
698             default:
699                 throw new IllegalStateException("Unreachable");
700         }
701     }
702
703     @Override

```

```

704     public long between(Temporal temporal1Inclusive, Temporal temporal2Exclusive) {
705         if (temporal1Inclusive.getClass() != temporal2Exclusive.getClass()) {
706             return temporal1Inclusive.until(temporal2Exclusive, this);
707         }
708         switch(this) {
709             case WEEK_BASED_YEARS:
710                 return Math.subtractExact(temporal2Exclusive.getLong(WEEK_BASED_YEAR),
711                     temporal1Inclusive.getLong(WEEK_BASED_YEAR));
712             case QUARTER_YEARS:
713                 return temporal1Inclusive.until(temporal2Exclusive, MONTHS) / 3;
714             default:
715                 throw new IllegalStateException("Unreachable");
716         }
717     }
718
719     @Override
720     public String toString() {
721         return name;
722     }
723 }
724 }

```

7.4 JulianFields.java

Secuencia 115: java/time/temporal/JulianFields.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *

```

```
29  *
30  *
31  *
32  * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62 package java.time.temporal;
63
64 import static java.time.temporal.ChronoField.EPOCH_DAY;
65 import static java.time.temporal.ChronoUnit.DAYS;
66 import static java.time.temporal.ChronoUnit.FOREVER;
67
68 import java.time.DateTimeException;
69 import java.time.chrono.ChronoLocalDate;
70 import java.time.chrono.Chronology;
71 import java.time.format.ResolverStyle;
72 import java.util.Map;
73
74 /**
75  * A set of date fields that provide access to Julian Days.
76  * <p>
77  * The Julian Day is a standard way of expressing date and time commonly used in the scientific
78  *   community.
79  * It is expressed as a decimal number of whole days where days start at midday.
80  * This class represents variations on Julian Days that count whole days from midnight.
81  * <p>
```

```

81  * The fields are implemented relative to {@link ChronoField#EPOCH_DAY EPOCH_DAY}.
82  * The fields are supported, and can be queried and set if {@code EPOCH_DAY} is available.
83  * The fields work with all chronologies.
84  *
85  * @implSpec
86  * This is an immutable and thread-safe class.
87  *
88  * @since 1.8
89  */
90 public final class JulianFields {
91
92     /**
93      * The offset from Julian to EPOCH DAY.
94      */
95     private static final long JULIAN_DAY_OFFSET = 2440588L;
96
97     /**
98      * Julian Day field.
99      * <p>
100     * This is an integer-based version of the Julian Day Number.
101     * Julian Day is a well-known system that represents the count of whole days since day 0,
102     * which is defined to be January 1, 4713 BCE in the Julian calendar, and -4713-11-24 Gregorian
103     * .
104     * The field has "JulianDay" as 'name', and 'DAYS' as 'baseUnit'.
105     * The field always refers to the local date-time, ignoring the offset or zone.
106     * <p>
107     * For date-times, 'JULIAN_DAY.getFrom()' assumes the same value from
108     * midnight until just before the next midnight.
109     * When 'JULIAN_DAY.adjustInto()' is applied to a date-time, the time of day portion remains
110     * unaltered.
111     * 'JULIAN_DAY.adjustInto()' and 'JULIAN_DAY.getFrom()' only apply to {@code Temporal} objects
112     * that
113     * can be converted into {@link ChronoField#EPOCH_DAY}.
114     * An {@link UnsupportedTemporalTypeException} is thrown for any other type of object.
115     * <p>
116     * In the resolving phase of parsing, a date can be created from a Julian Day field.
117     * In {@linkplain ResolverStyle#STRICT strict mode} and {@linkplain ResolverStyle#SMART smart
118     * mode}
119     * the Julian Day value is validated against the range of valid values.
120     * In {@linkplain ResolverStyle#LENIENT lenient mode} no validation occurs.
121     *
122     * <h3>Astronomical and Scientific Notes</h3>
123     * The standard astronomical definition uses a fraction to indicate the time-of-day,
124     * thus 3.25 would represent the time 18:00, since days start at midday.
125     * This implementation uses an integer and days starting at midnight.
126     * The integer value for the Julian Day Number is the astronomical Julian Day value at midday
127     * of the date in question.
128     * This amounts to the astronomical Julian Day, rounded to an integer {@code JDN = floor(JD +
129     * 0.5)}.
130     *
131     * <pre>
132     * | ISO date          | Julian Day Number | Astronomical Julian Day |
133     * | 1970-01-01T00:00 |         2,440,588 |         2,440,587.5     |

```



```

129 * | 1970-01-01T06:00 |      2,440,588 |      2,440,587.75 |
130 * | 1970-01-01T12:00 |      2,440,588 |      2,440,588.0   |
131 * | 1970-01-01T18:00 |      2,440,588 |      2,440,588.25 |
132 * | 1970-01-02T00:00 |      2,440,589 |      2,440,588.5   |
133 * | 1970-01-02T06:00 |      2,440,589 |      2,440,588.75 |
134 * | 1970-01-02T12:00 |      2,440,589 |      2,440,589.0   |
135 * </pre>
136 * <p>
137 * Julian Days are sometimes taken to imply Universal Time or UTC, but this
138 * implementation always uses the Julian Day number for the local date,
139 * regardless of the offset or time-zone.
140 */
141 public static final TemporalField JULIAN_DAY = Field.JULIAN_DAY;
142
143 /**
144  * Modified Julian Day field.
145  * <p>
146  * This is an integer-based version of the Modified Julian Day Number.
147  * Modified Julian Day (MJD) is a well-known system that counts days continuously.
148  * It is defined relative to astronomical Julian Day as {@code MJD = JD - 2400000.5}.
149  * Each Modified Julian Day runs from midnight to midnight.
150  * The field always refers to the local date-time, ignoring the offset or zone.
151  * <p>
152  * For date-times, 'MODIFIED_JULIAN_DAY.getFrom()' assumes the same value from
153  * midnight until just before the next midnight.
154  * When 'MODIFIED_JULIAN_DAY.adjustInto()' is applied to a date-time, the time of day portion
155  * remains unaltered.
156  * 'MODIFIED_JULIAN_DAY.adjustInto()' and 'MODIFIED_JULIAN_DAY.getFrom()' only apply to {@code
157  * Temporal} objects
158  * that can be converted into {@link ChronoField#EPOCH_DAY}.
159  * An {@link UnsupportedOperationException} is thrown for any other type of object.
160  * <p>
161  * This implementation is an integer version of MJD with the decimal part rounded to floor.
162  * <p>
163  * In the resolving phase of parsing, a date can be created from a Modified Julian Day field.
164  * In {@linkplain ResolverStyle#STRICT strict mode} and {@linkplain ResolverStyle#SMART smart
165  * mode}
166  * the Modified Julian Day value is validated against the range of valid values.
167  * In {@linkplain ResolverStyle#LENIENT lenient mode} no validation occurs.
168  *
169  * <h3>Astronomical and Scientific Notes</h3>
170  * <pre>
171  * | ISO date          | Modified Julian Day |      Decimal MJD |
172  * | 1970-01-01T00:00 |          40,587     |      40,587.0   |
173  * | 1970-01-01T06:00 |          40,587     |      40,587.25  |
174  * | 1970-01-01T12:00 |          40,587     |      40,587.5   |
175  * | 1970-01-01T18:00 |          40,587     |      40,587.75  |
176  * | 1970-01-02T00:00 |          40,588     |      40,588.0   |
177  * | 1970-01-02T06:00 |          40,588     |      40,588.25  |
178  * | 1970-01-02T12:00 |          40,588     |      40,588.5   |
179  * </pre>
180  *
181  * Modified Julian Days are sometimes taken to imply Universal Time or UTC, but this

```

```

179     * implementation always uses the Modified Julian Day for the local date,
180     * regardless of the offset or time-zone.
181     */
182     public static final TemporalField MODIFIED_JULIAN_DAY = Field.MODIFIED_JULIAN_DAY;
183
184     /**
185     * Rata Die field.
186     * <p>
187     * Rata Die counts whole days continuously starting day 1 at midnight at the beginning of
188     * 0001-01-01 (ISO).
189     * The field always refers to the local date-time, ignoring the offset or zone.
190     * <p>
191     * For date-times, 'RATA_DIE.getFrom()' assumes the same value from
192     * midnight until just before the next midnight.
193     * When 'RATA_DIE.adjustInto()' is applied to a date-time, the time of day portion remains
194     * unaltered.
195     * 'RATA_DIE.adjustInto()' and 'RATA_DIE.getFrom()' only apply to {@code Temporal} objects
196     * that can be converted into {@link ChronoField#EPOCH_DAY}.
197     * An {@link UnsupportedOperationException} is thrown for any other type of object.
198     * <p>
199     * In the resolving phase of parsing, a date can be created from a Rata Die field.
200     * In {@linkplain ResolverStyle#STRICT strict mode} and {@linkplain ResolverStyle#SMART smart
201     * mode}
202     * the Rata Die value is validated against the range of valid values.
203     * In {@linkplain ResolverStyle#LENIENT lenient mode} no validation occurs.
204     */
205     public static final TemporalField RATA_DIE = Field.RATA_DIE;
206
207     /**
208     * Restricted constructor.
209     */
210     private JulianFields() {
211         throw new AssertionError("Not instantiable");
212     }
213
214     /**
215     * Implementation of JulianFields. Each instance is a singleton.
216     */
217     private static enum Field implements TemporalField {
218         JULIAN_DAY("JulianDay", DAYS, FOREVER, JULIAN_DAY_OFFSET),
219         MODIFIED_JULIAN_DAY("ModifiedJulianDay", DAYS, FOREVER, 40587L),
220         RATA_DIE("RataDie", DAYS, FOREVER, 719163L);
221
222         private static final long serialVersionUID = -7501623920830201812L;
223
224         private final transient String name;
225         private final transient TemporalUnit baseUnit;
226         private final transient TemporalUnit rangeUnit;
227         private final transient ValueRange range;
228         private final transient long offset;
229
230         private Field(String name, TemporalUnit baseUnit, TemporalUnit rangeUnit, long offset) {
231             this.name = name;

```

```
229         this.baseUnit = baseUnit;
230         this.rangeUnit = rangeUnit;
231         this.range = ValueRange.of(-365243219162L + offset, 365241780471L + offset);
232         this.offset = offset;
233     }
234
235     //-----
236     @Override
237     public TemporalUnit getBaseUnit() {
238         return baseUnit;
239     }
240
241     @Override
242     public TemporalUnit getRangeUnit() {
243         return rangeUnit;
244     }
245
246     @Override
247     public boolean isDateBased() {
248         return true;
249     }
250
251     @Override
252     public boolean isTimeBased() {
253         return false;
254     }
255
256     @Override
257     public ValueRange range() {
258         return range;
259     }
260
261     //-----
262     @Override
263     public boolean isSupportedBy(TemporalAccessor temporal) {
264         return temporal.isSupported(EPOCH_DAY);
265     }
266
267     @Override
268     public ValueRange rangeRefinedBy(TemporalAccessor temporal) {
269         if (isSupportedBy(temporal) == false) {
270             throw new DateTimeException("Unsupported field: " + this);
271         }
272         return range();
273     }
274
275     @Override
276     public long getFrom(TemporalAccessor temporal) {
277         return temporal.getLong(EPOCH_DAY) + offset;
278     }
279
280     @SuppressWarnings("unchecked")
281     @Override
```

```

282     public <R extends Temporal> R adjustInto(R temporal, long newValue) {
283         if (range().isValidValue(newValue) == false) {
284             throw new DateTimeException("Invalid value: " + name + " " + newValue);
285         }
286         return (R) temporal.with(EPOCH_DAY, Math.subtractExact(newValue, offset));
287     }
288
289     //-----
290     @Override
291     public ChronoLocalDate resolve(
292         Map<TemporalField, Long> fieldValues, TemporalAccessor partialTemporal,
293         ResolverStyle resolverStyle) {
294         long value = fieldValues.remove(this);
295         Chronology chrono = Chronology.from(partialTemporal);
296         if (resolverStyle == ResolverStyle.LENIENT) {
297             return chrono.dateEpochDay(Math.subtractExact(value, offset));
298         }
299         range().checkValidValue(value, this);
300         return chrono.dateEpochDay(value - offset);
301     }
302
303     //-----
304     @Override
305     public String toString() {
306         return name;
307     }
308 }

```

7.5 Temporal.java

Secuencia 116: java/time/temporal/Temporal.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *

```

```
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.temporal;
63
64  import java.time.DateTimeException;
65
66  /**
67   * Framework-level interface defining read-write access to a temporal object,
68   * such as a date, time, offset or some combination of these.
69   * <p>
70   * This is the base interface type for date, time and offset objects that
71   * are complete enough to be manipulated using plus and minus.
72   * It is implemented by those classes that can provide and manipulate information
73   * as {@linkplain TemporalField fields} or {@linkplain TemporalQuery queries}.
74   * See {@link TemporalAccessor} for the read-only version of this interface.
```

```

75 * <p>
76 * Most date and time information can be represented as a number.
77 * These are modeled using {@code TemporalField} with the number held using
78 * a {@code long} to handle large values. Year, month and day-of-month are
79 * simple examples of fields, but they also include instant and offsets.
80 * See {@link ChronoField} for the standard set of fields.
81 * <p>
82 * Two pieces of date/time information cannot be represented by numbers,
83 * the {@linkplain java.time.chrono.Chronology chronology} and the
84 * {@linkplain java.time.ZoneId time-zone}.
85 * These can be accessed via {@link #query(TemporalQuery) queries} using
86 * the static methods defined on {@link TemporalQuery}.
87 * <p>
88 * This interface is a framework-level interface that should not be widely
89 * used in application code. Instead, applications should create and pass
90 * around instances of concrete types, such as {@code LocalDate}.
91 * There are many reasons for this, part of which is that implementations
92 * of this interface may be in calendar systems other than ISO.
93 * See {@link java.time.chrono.ChronoLocalDate} for a fuller discussion of the issues.
94 *
95 * <h3>When to implement</h3>
96 * <p>
97 * A class should implement this interface if it meets three criteria:
98 * <ul>
99 * <li>it provides access to date/time/offset information, as per {@code TemporalAccessor}
100 * <li>the set of fields are contiguous from the largest to the smallest
101 * <li>the set of fields are complete, such that no other field is needed to define the
102 *   valid range of values for the fields that are represented
103 * </ul>
104 * <p>
105 * Four examples make this clear:
106 * <ul>
107 * <li>{@code LocalDate} implements this interface as it represents a set of fields
108 *   that are contiguous from days to forever and require no external information to determine
109 *   the validity of each date. It is therefore able to implement plus/minus correctly.
110 * <li>{@code LocalTime} implements this interface as it represents a set of fields
111 *   that are contiguous from nanos to within days and require no external information to determine
112 *   validity. It is able to implement plus/minus correctly, by wrapping around the day.
113 * <li>{@code MonthDay}, the combination of month-of-year and day-of-month, does not implement
114 *   this interface. While the combination is contiguous, from days to months within years,
115 *   the combination does not have sufficient information to define the valid range of values
116 *   for day-of-month. As such, it is unable to implement plus/minus correctly.
117 * <li>The combination day-of-week and day-of-month ("Friday the 13th") should not implement
118 *   this interface. It does not represent a contiguous set of fields, as days to weeks overlaps
119 *   days to months.
120 * </ul>
121 *
122 * @implSpec
123 * This interface places no restrictions on the mutability of implementations,
124 * however immutability is strongly recommended.
125 * All implementations must be {@link Comparable}.
126 *
127 * @since 1.8

```

```

128 */
129 public interface Temporal extends TemporalAccessor {
130
131     /**
132      * Checks if the specified unit is supported.
133      * <p>
134      * This checks if the specified unit can be added to, or subtracted from, this date-time.
135      * If false, then calling the {@link #plus(long, TemporalUnit)} and
136      * {@link #minus(long, TemporalUnit) minus} methods will throw an exception.
137      *
138      * @implSpec
139      * Implementations must check and handle all units defined in {@link ChronoUnit}.
140      * If the unit is supported, then true must be returned, otherwise false must be returned.
141      * <p>
142      * If the field is not a {@code ChronoUnit}, then the result of this method
143      * is obtained by invoking {@code TemporalUnit.isSupportedBy(Temporal)}
144      * passing {@code this} as the argument.
145      * <p>
146      * Implementations must ensure that no observable state is altered when this
147      * read-only method is invoked.
148      *
149      * @param unit the unit to check, null returns false
150      * @return true if the unit can be added/subtracted, false if not
151      */
152     boolean isSupported(TemporalUnit unit);
153
154     /**
155      * Returns an adjusted object of the same type as this object with the adjustment made.
156      * <p>
157      * This adjusts this date-time according to the rules of the specified adjuster.
158      * A simple adjuster might simply set the one of the fields, such as the year field.
159      * A more complex adjuster might set the date to the last day of the month.
160      * A selection of common adjustments is provided in
161      * {@link java.time.temporal.TemporalAdjusters TemporalAdjusters}.
162      * These include finding the "last day of the month" and "next Wednesday".
163      * The adjuster is responsible for handling special cases, such as the varying
164      * lengths of month and leap years.
165      * <p>
166      * Some example code indicating how and why this method is used:
167      * <pre>
168      * date = date.with(Month.JULY);           // most key classes implement TemporalAdjuster
169      * date = date.with(lastDayOfMonth());    // static import from Adjusters
170      * date = date.with(next(WEDNESDAY));     // static import from Adjusters and DayOfWeek
171      * </pre>
172      *
173      * @implSpec
174      * <p>
175      * Implementations must not alter either this object or the specified temporal object.
176      * Instead, an adjusted copy of the original must be returned.
177      * This provides equivalent, safe behavior for immutable and mutable implementations.
178      * <p>
179      * The default implementation must behave equivalent to this code:
180      * <pre>

```

```

181     * return adjuster.adjustInto(this);
182     * </pre>
183     *
184     * @param adjuster the adjuster to use, not null
185     * @return an object of the same type with the specified adjustment made, not null
186     * @throws DateTimeException if unable to make the adjustment
187     * @throws ArithmeticException if numeric overflow occurs
188     */
189     default Temporal with(TemporalAdjuster adjuster) {
190         return adjuster.adjustInto(this);
191     }
192
193     /**
194     * Returns an object of the same type as this object with the specified field altered.
195     * <p>
196     * This returns a new object based on this one with the value for the specified field changed.
197     * For example, on a {@code LocalDate}, this could be used to set the year, month or day-of-
198     * month.
199     * The returned object will have the same observable type as this object.
200     * <p>
201     * In some cases, changing a field is not fully defined. For example, if the target object is
202     * a date representing the 31st January, then changing the month to February would be unclear.
203     * In cases like this, the field is responsible for resolving the result. Typically it will
204     * choose
205     * the previous valid date, which would be the last valid day of February in this example.
206     *
207     * @implSpec
208     * Implementations must check and handle all fields defined in {@link ChronoField}.
209     * If the field is supported, then the adjustment must be performed.
210     * If unsupported, then an {@code UnsupportedOperationException} must be thrown.
211     * <p>
212     * If the field is not a {@code ChronoField}, then the result of this method
213     * is obtained by invoking {@code TemporalField.adjustInto(Temporal, long)}
214     * passing {@code this} as the first argument.
215     * <p>
216     * Implementations must not alter this object.
217     * Instead, an adjusted copy of the original must be returned.
218     * This provides equivalent, safe behavior for immutable and mutable implementations.
219     *
220     * @param field the field to set in the result, not null
221     * @param newValue the new value of the field in the result
222     * @return an object of the same type with the specified field set, not null
223     * @throws DateTimeException if the field cannot be set
224     * @throws UnsupportedOperationException if the field is not supported
225     * @throws ArithmeticException if numeric overflow occurs
226     */
227     Temporal with(TemporalField field, long newValue);
228
229     /**
230     * Returns an object of the same type as this object with an amount added.
231     * <p>
232     * This adjusts this temporal, adding according to the rules of the specified amount.

```



```

232 * The amount is typically a {@link java.time.Period} but may be any other type implementing
233 * the {@link TemporalAmount} interface, such as {@link java.time.Duration}.
234 * <p>
235 * Some example code indicating how and why this method is used:
236 * <pre>
237 *   date = date.plus(period);           // add a Period instance
238 *   date = date.plus(duration);         // add a Duration instance
239 *   date = date.plus(workingDays(6));    // example user-written workingDays method
240 * </pre>
241 * <p>
242 * Note that calling {@code plus} followed by {@code minus} is not guaranteed to
243 * return the same date-time.
244 *
245 * @implSpec
246 * <p>
247 * Implementations must not alter either this object or the specified temporal object.
248 * Instead, an adjusted copy of the original must be returned.
249 * This provides equivalent, safe behavior for immutable and mutable implementations.
250 * <p>
251 * The default implementation must behave equivalent to this code:
252 * <pre>
253 *   return amount.addTo(this);
254 * </pre>
255 *
256 * @param amount the amount to add, not null
257 * @return an object of the same type with the specified adjustment made, not null
258 * @throws DateTimeException if the addition cannot be made
259 * @throws ArithmeticException if numeric overflow occurs
260 */
261 default Temporal plus(TemporalAmount amount) {
262     return amount.addTo(this);
263 }
264
265 /**
266 * Returns an object of the same type as this object with the specified period added.
267 * <p>
268 * This method returns a new object based on this one with the specified period added.
269 * For example, on a {@code LocalDate}, this could be used to add a number of years, months or
270 *   days.
271 * The returned object will have the same observable type as this object.
272 * <p>
273 * In some cases, changing a field is not fully defined. For example, if the target object is
274 * a date representing the 31st January, then adding one month would be unclear.
275 * In cases like this, the field is responsible for resolving the result. Typically it will
276 *   choose
277 * the previous valid date, which would be the last valid day of February in this example.
278 *
279 * @implSpec
280 * Implementations must check and handle all units defined in {@link ChronoUnit}.
281 * If the unit is supported, then the addition must be performed.
282 * If unsupported, then an {@code UnsupportedTemporalTypeException} must be thrown.
283 * <p>
284 * If the unit is not a {@code ChronoUnit}, then the result of this method

```

```

283 * is obtained by invoking {@code TemporalUnit.addTo(Temporal, long)}
284 * passing {@code this} as the first argument.
285 * <p>
286 * Implementations must not alter this object.
287 * Instead, an adjusted copy of the original must be returned.
288 * This provides equivalent, safe behavior for immutable and mutable implementations.
289 *
290 * @param amountToAdd the amount of the specified unit to add, may be negative
291 * @param unit the unit of the amount to add, not null
292 * @return an object of the same type with the specified period added, not null
293 * @throws DateTimeException if the unit cannot be added
294 * @throws UnsupportedTemporalTypeException if the unit is not supported
295 * @throws ArithmeticException if numeric overflow occurs
296 */
297 Temporal plus(long amountToAdd, TemporalUnit unit);
298
299 //-----
300 /**
301 * Returns an object of the same type as this object with an amount subtracted.
302 * <p>
303 * This adjusts this temporal, subtracting according to the rules of the specified amount.
304 * The amount is typically a {@link java.time.Period} but may be any other type implementing
305 * the {@link TemporalAmount} interface, such as {@link java.time.Duration}.
306 * <p>
307 * Some example code indicating how and why this method is used:
308 * <pre>
309 * date = date.minus(period);           // subtract a Period instance
310 * date = date.minus(duration);         // subtract a Duration instance
311 * date = date.minus(workingDays(6));   // example user-written workingDays method
312 * </pre>
313 * <p>
314 * Note that calling {@code plus} followed by {@code minus} is not guaranteed to
315 * return the same date-time.
316 *
317 * @implSpec
318 * <p>
319 * Implementations must not alter either this object or the specified temporal object.
320 * Instead, an adjusted copy of the original must be returned.
321 * This provides equivalent, safe behavior for immutable and mutable implementations.
322 * <p>
323 * The default implementation must behave equivalent to this code:
324 * <pre>
325 * return amount.subtractFrom(this);
326 * </pre>
327 *
328 * @param amount the amount to subtract, not null
329 * @return an object of the same type with the specified adjustment made, not null
330 * @throws DateTimeException if the subtraction cannot be made
331 * @throws ArithmeticException if numeric overflow occurs
332 */
333 default Temporal minus(TemporalAmount amount) {
334     return amount.subtractFrom(this);
335 }

```

```

336
337 /**
338  * Returns an object of the same type as this object with the specified period subtracted.
339  * <p>
340  * This method returns a new object based on this one with the specified period subtracted.
341  * For example, on a {@code LocalDate}, this could be used to subtract a number of years,
342     months or days.
343  * The returned object will have the same observable type as this object.
344  * <p>
345  * In some cases, changing a field is not fully defined. For example, if the target object is
346  * a date representing the 31st March, then subtracting one month would be unclear.
347  * In cases like this, the field is responsible for resolving the result. Typically it will
348     choose
349  * the previous valid date, which would be the last valid day of February in this example.
350  *
351  * @implSpec
352  * Implementations must behave in a manor equivalent to the default method behavior.
353  * <p>
354  * Implementations must not alter this object.
355  * Instead, an adjusted copy of the original must be returned.
356  * This provides equivalent, safe behavior for immutable and mutable implementations.
357  * <p>
358  * The default implementation must behave equivalent to this code:
359  * <pre>
360  *   return (amountToSubtract == Long.MIN_VALUE ?
361     plus(Long.MAX_VALUE, unit).plus(1, unit) : plus(-amountToSubtract, unit));
362  * </pre>
363  *
364  * @param amountToSubtract the amount of the specified unit to subtract, may be negative
365  * @param unit the unit of the amount to subtract, not null
366  * @return an object of the same type with the specified period subtracted, not null
367  * @throws DateTimeException if the unit cannot be subtracted
368  * @throws UnsupportedTemporalTypeException if the unit is not supported
369  * @throws ArithmeticException if numeric overflow occurs
370  */
371 default Temporal minus(long amountToSubtract, TemporalUnit unit) {
372     return (amountToSubtract == Long.MIN_VALUE ? plus(Long.MAX_VALUE, unit).plus(1, unit) :
373     plus(-amountToSubtract, unit));
374 }
375
376 //-----
377 /**
378  * Calculates the amount of time until another temporal in terms of the specified unit.
379  * <p>
380  * This calculates the amount of time between two temporal objects
381  * in terms of a single {@code TemporalUnit}.
382  * The start and end points are {@code this} and the specified temporal.
383  * The end point is converted to be of the same type as the start point if different.
384  * The result will be negative if the end is before the start.
385  * For example, the amount in hours between two temporal objects can be
386  * calculated using {@code startTime.until(endTime, HOURS)}.
387  * <p>
388  * The calculation returns a whole number, representing the number of

```

```

386 * complete units between the two temporals.
387 * For example, the amount in hours between the times 11:30 and 13:29
388 * will only be one hour as it is one minute short of two hours.
389 * <p>
390 * There are two equivalent ways of using this method.
391 * The first is to invoke this method directly.
392 * The second is to use {@link TemporalUnit#between(Temporal, Temporal)}:
393 * <pre>
394 * // these two lines are equivalent
395 * temporal = start.until(end, unit);
396 * temporal = unit.between(start, end);
397 * </pre>
398 * The choice should be made based on which makes the code more readable.
399 * <p>
400 * For example, this method allows the number of days between two dates to
401 * be calculated:
402 * <pre>
403 * long daysBetween = start.until(end, DAYS);
404 * // or alternatively
405 * long daysBetween = DAYS.between(start, end);
406 * </pre>
407 *
408 * @implSpec
409 * Implementations must begin by checking to ensure that the input temporal
410 * object is of the same observable type as the implementation.
411 * They must then perform the calculation for all instances of {@link ChronoUnit}.
412 * An {@code UnsupportedOperationException} must be thrown for {@code ChronoUnit}
413 * instances that are unsupported.
414 * <p>
415 * If the unit is not a {@code ChronoUnit}, then the result of this method
416 * is obtained by invoking {@code TemporalUnit.between(Temporal, Temporal)}
417 * passing {@code this} as the first argument and the converted input temporal as
418 * the second argument.
419 * <p>
420 * In summary, implementations must behave in a manner equivalent to this pseudo-code:
421 * <pre>
422 * // convert the end temporal to the same type as this class
423 * if (unit instanceof ChronoUnit) {
424 * // if unit is supported, then calculate and return result
425 * // else throw UnsupportedOperationException for unsupported units
426 * }
427 * return unit.between(this, convertedEndTemporal);
428 * </pre>
429 * <p>
430 * Note that the unit's {@code between} method must only be invoked if the
431 * two temporal objects have exactly the same type evaluated by {@code getClass()}.
432 * <p>
433 * Implementations must ensure that no observable state is altered when this
434 * read-only method is invoked.
435 *
436 * @param endExclusive the end temporal, exclusive, converted to be of the
437 * same type as this object, not null
438 * @param unit the unit to measure the amount in, not null

```

```
439     * @return the amount of time between this temporal object and the specified one
440     * in terms of the unit; positive if the specified object is later than this one,
441     * negative if it is earlier than this one
442     * @throws DateTimeException if the amount cannot be calculated, or the end
443     * temporal cannot be converted to the same type as this temporal
444     * @throws UnsupportedTemporalTypeException if the unit is not supported
445     * @throws ArithmeticException if numeric overflow occurs
446     */
447     long until(Temporal endExclusive, TemporalUnit unit);
448
449 }
```

7.6 TemporalAccessor.java

Secuencia 117: java/time/temporal/TemporalAccessor.java

```
1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
```

```

39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62 package java.time.temporal;
63
64 import java.time.DateTimeException;
65 import java.util.Objects;
66
67 /**
68  * Framework-level interface defining read-only access to a temporal object,
69  * such as a date, time, offset or some combination of these.
70  * <p>
71  * This is the base interface type for date, time and offset objects.
72  * It is implemented by those classes that can provide information
73  * as {@linkplain TemporalField fields} or {@linkplain TemporalQuery queries}.
74  * <p>
75  * Most date and time information can be represented as a number.
76  * These are modeled using {@code TemporalField} with the number held using
77  * a {@code long} to handle large values. Year, month and day-of-month are
78  * simple examples of fields, but they also include instant and offsets.
79  * See {@link ChronoField} for the standard set of fields.
80  * <p>
81  * Two pieces of date/time information cannot be represented by numbers,
82  * the {@linkplain java.time.chrono.Chronology chronology} and the
83  * {@linkplain java.time.ZoneId time-zone}.
84  * These can be accessed via {@linkplain #query(TemporalQuery) queries} using
85  * the static methods defined on {@link TemporalQuery}.
86  * <p>
87  * A sub-interface, {@link Temporal}, extends this definition to one that also
88  * supports adjustment and manipulation on more complete temporal objects.
89  * <p>
90  * This interface is a framework-level interface that should not be widely
91  * used in application code. Instead, applications should create and pass

```

```

92  * around instances of concrete types, such as {@code LocalDate}.
93  * There are many reasons for this, part of which is that implementations
94  * of this interface may be in calendar systems other than ISO.
95  * See {@link java.time.chrono.ChronoLocalDate} for a fuller discussion of the issues.
96  *
97  * @implSpec
98  * This interface places no restrictions on the mutability of implementations,
99  * however immutability is strongly recommended.
100 *
101 * @since 1.8
102 */
103 public interface TemporalAccessor {
104
105     /**
106      * Checks if the specified field is supported.
107      * <p>
108      * This checks if the date-time can be queried for the specified field.
109      * If false, then calling the {@link #range(TemporalField) range} and {@link #get(TemporalField) get}
110      * methods will throw an exception.
111      *
112      * @implSpec
113      * Implementations must check and handle all fields defined in {@link ChronoField}.
114      * If the field is supported, then true must be returned, otherwise false must be returned.
115      * <p>
116      * If the field is not a {@code ChronoField}, then the result of this method
117      * is obtained by invoking {@code TemporalField.isSupportedBy(TemporalAccessor)}
118      * passing {@code this} as the argument.
119      * <p>
120      * Implementations must ensure that no observable state is altered when this
121      * read-only method is invoked.
122      *
123      * @param field the field to check, null returns false
124      * @return true if this date-time can be queried for the field, false if not
125      */
126     boolean isSupported(TemporalField field);
127
128     /**
129      * Gets the range of valid values for the specified field.
130      * <p>
131      * All fields can be expressed as a {@code long} integer.
132      * This method returns an object that describes the valid range for that value.
133      * The value of this temporal object is used to enhance the accuracy of the returned range.
134      * If the date-time cannot return the range, because the field is unsupported or for
135      * some other reason, an exception will be thrown.
136      * <p>
137      * Note that the result only describes the minimum and maximum valid values
138      * and it is important not to read too much into them. For example, there
139      * could be values within the range that are invalid for the field.
140      *
141      * @implSpec
142      * Implementations must check and handle all fields defined in {@link ChronoField}.
143      * If the field is supported, then the range of the field must be returned.

```

```

144 * If unsupported, then an {@code UnsupportedOperationException} must be thrown.
145 * <p>
146 * If the field is not a {@code ChronoField}, then the result of this method
147 * is obtained by invoking {@code TemporalField.rangeRefinedBy(TemporalAccessor1)}
148 * passing {@code this} as the argument.
149 * <p>
150 * Implementations must ensure that no observable state is altered when this
151 * read-only method is invoked.
152 * <p>
153 * The default implementation must behave equivalent to this code:
154 * <pre>
155 *   if (field instanceof ChronoField) {
156 *       if (isSupported(field)) {
157 *           return field.range();
158 *       }
159 *       throw new UnsupportedOperationException("Unsupported field: " + field);
160 *   }
161 *   return field.rangeRefinedBy(this);
162 * </pre>
163 *
164 * @param field the field to query the range for, not null
165 * @return the range of valid values for the field, not null
166 * @throws DateTimeException if the range for the field cannot be obtained
167 * @throws UnsupportedOperationException if the field is not supported
168 */
169 default ValueRange range(TemporalField field) {
170     if (field instanceof ChronoField) {
171         if (isSupported(field)) {
172             return field.range();
173         }
174         throw new UnsupportedOperationException("Unsupported field: " + field);
175     }
176     Objects.requireNonNull(field, "field");
177     return field.rangeRefinedBy(this);
178 }
179
180 /**
181 * Gets the value of the specified field as an {@code int}.
182 * <p>
183 * This queries the date-time for the value of the specified field.
184 * The returned value will always be within the valid range of values for the field.
185 * If the date-time cannot return the value, because the field is unsupported or for
186 * some other reason, an exception will be thrown.
187 *
188 * @implSpec
189 * Implementations must check and handle all fields defined in {@link ChronoField}.
190 * If the field is supported and has an {@code int} range, then the value of
191 * the field must be returned.
192 * If unsupported, then an {@code UnsupportedOperationException} must be thrown.
193 * <p>
194 * If the field is not a {@code ChronoField}, then the result of this method
195 * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
196 * passing {@code this} as the argument.

```



```

197 * <p>
198 * Implementations must ensure that no observable state is altered when this
199 * read-only method is invoked.
200 * <p>
201 * The default implementation must behave equivalent to this code:
202 * <pre>
203 *   if (range(field).isIntValue()) {
204 *       return range(field).checkValidIntValue(getLong(field), field);
205 *   }
206 *   throw new UnsupportedOperationException("Invalid field " + field + " for get() method,
           use getLong() instead");
207 * </pre>
208 *
209 * @param field the field to get, not null
210 * @return the value for the field, within the valid range of values
211 * @throws DateTimeException if a value for the field cannot be obtained or
212 *         the value is outside the range of valid values for the field
213 * @throws UnsupportedOperationException if the field is not supported or
214 *         the range of values exceeds an {@code int}
215 * @throws ArithmeticException if numeric overflow occurs
216 */
217 default int get(TemporalField field) {
218     ValueRange range = range(field);
219     if (range.isIntValue() == false) {
220         throw new UnsupportedOperationException("Invalid field " + field + " for get()
           method, use getLong() instead");
221     }
222     long value = getLong(field);
223     if (range.isValidValue(value) == false) {
224         throw new DateTimeException("Invalid value for " + field + " (valid values " + range +
           "): " + value);
225     }
226     return (int) value;
227 }
228
229 /**
230 * Gets the value of the specified field as a {@code long}.
231 * <p>
232 * This queries the date-time for the value of the specified field.
233 * The returned value may be outside the valid range of values for the field.
234 * If the date-time cannot return the value, because the field is unsupported or for
235 * some other reason, an exception will be thrown.
236 *
237 * @implSpec
238 * Implementations must check and handle all fields defined in {@link ChronoField}.
239 * If the field is supported, then the value of the field must be returned.
240 * If unsupported, then an {@code UnsupportedOperationException} must be thrown.
241 * <p>
242 * If the field is not a {@code ChronoField}, then the result of this method
243 * is obtained by invoking {@code TemporalField.getFrom(TemporalAccessor)}
244 * passing {@code this} as the argument.
245 * <p>
246 * Implementations must ensure that no observable state is altered when this

```

```

247     * read-only method is invoked.
248     *
249     * @param field the field to get, not null
250     * @return the value for the field
251     * @throws DateTimeException if a value for the field cannot be obtained
252     * @throws UnsupportedTemporalTypeException if the field is not supported
253     * @throws ArithmeticException if numeric overflow occurs
254     */
255     long getLong(TemporalField field);
256
257     /**
258     * Queries this date-time.
259     * <p>
260     * This queries this date-time using the specified query strategy object.
261     * <p>
262     * Queries are a key tool for extracting information from date-times.
263     * They exists to externalize the process of querying, permitting different
264     * approaches, as per the strategy design pattern.
265     * Examples might be a query that checks if the date is the day before February 29th
266     * in a leap year, or calculates the number of days to your next birthday.
267     * <p>
268     * The most common query implementations are method references, such as
269     * {@code LocalDate::from} and {@code ZoneId::from}.
270     * Additional implementations are provided as static methods on {@link TemporalQuery}.
271     *
272     * @implSpec
273     * The default implementation must behave equivalent to this code:
274     * <pre>
275     * if (query == TemporalQueries.zoneId() ||
276     *     query == TemporalQueries.chronology() || query == TemporalQueries.precision()) {
277     *     return null;
278     * }
279     * return query.queryFrom(this);
280     * </pre>
281     * Future versions are permitted to add further queries to the if statement.
282     * <p>
283     * All classes implementing this interface and overriding this method must call
284     * {@code TemporalAccessor.super.query(query)}. JDK classes may avoid calling
285     * super if they provide behavior equivalent to the default behaviour, however
286     * non-JDK classes may not utilize this optimization and must call {@code super}.
287     * <p>
288     * If the implementation can supply a value for one of the queries listed in the
289     * if statement of the default implementation, then it must do so.
290     * For example, an application-defined {@code HourMin} class storing the hour
291     * and minute must override this method as follows:
292     * <pre>
293     * if (query == TemporalQueries.precision()) {
294     *     return MINUTES;
295     * }
296     * return TemporalAccessor.super.query(query);
297     * </pre>
298     * <p>
299     * Implementations must ensure that no observable state is altered when this

```

```

300     * read-only method is invoked.
301     *
302     * @param <R> the type of the result
303     * @param query the query to invoke, not null
304     * @return the query result, null may be returned (defined by the query)
305     * @throws DateTimeException if unable to query
306     * @throws ArithmeticException if numeric overflow occurs
307     */
308     default <R> R query(TemporalQuery<R> query) {
309         if (query == TemporalQueries.zoneId()
310             || query == TemporalQueries.chronology()
311             || query == TemporalQueries.precision()) {
312             return null;
313         }
314         return query.queryFrom(this);
315     }
316
317 }

```

7.7 TemporalAdjuster.java

Secuencia 118: java/time/temporal/TemporalAdjuster.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *

```

```
32 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
62 package java.time.temporal;
63
64 import java.time.DateTimeException;
65
66 /**
67  * Strategy for adjusting a temporal object.
68  * <p>
69  * Adjusters are a key tool for modifying temporal objects.
70  * They exist to externalize the process of adjustment, permitting different
71  * approaches, as per the strategy design pattern.
72  * Examples might be an adjuster that sets the date avoiding weekends, or one that
73  * sets the date to the last day of the month.
74  * <p>
75  * There are two equivalent ways of using a {@code TemporalAdjuster}.
76  * The first is to invoke the method on this interface directly.
77  * The second is to use {@link Temporal#with(TemporalAdjuster)}:
78  * <pre>
79  *     // these two lines are equivalent, but the second approach is recommended
80  *     temporal = thisAdjuster.adjustInto(temporal);
81  *     temporal = temporal.with(thisAdjuster);
82  * </pre>
83  * It is recommended to use the second approach, {@code with(TemporalAdjuster)},
84  * as it is a lot clearer to read in code.
```

```

85  * <p>
86  * The {@link TemporalAdjusters} class contains a standard set of adjusters,
87  * available as static methods.
88  * These include:
89  * <ul>
90  * <li>finding the first or last day of the month
91  * <li>finding the first day of next month
92  * <li>finding the first or last day of the year
93  * <li>finding the first day of next year
94  * <li>finding the first or last day-of-week within a month, such as "first Wednesday in June"
95  * <li>finding the next or previous day-of-week, such as "next Thursday"
96  * </ul>
97  *
98  * @implSpec
99  * This interface places no restrictions on the mutability of implementations,
100  * however immutability is strongly recommended.
101  *
102  * @see TemporalAdjusters
103  * @since 1.8
104  */
105  @FunctionalInterface
106  public interface TemporalAdjuster {
107
108      /**
109       * Adjusts the specified temporal object.
110       * <p>
111       * This adjusts the specified temporal object using the logic
112       * encapsulated in the implementing class.
113       * Examples might be an adjuster that sets the date avoiding weekends, or one that
114       * sets the date to the last day of the month.
115       * <p>
116       * There are two equivalent ways of using this method.
117       * The first is to invoke this method directly.
118       * The second is to use {@link Temporal#with(TemporalAdjuster)}:
119       * <pre>
120       *     // these two lines are equivalent, but the second approach is recommended
121       *     temporal = thisAdjuster.adjustInto(temporal);
122       *     temporal = temporal.with(thisAdjuster);
123       * </pre>
124       * It is recommended to use the second approach, {@code with(TemporalAdjuster)},
125       * as it is a lot clearer to read in code.
126       *
127       * @implSpec
128       * The implementation must take the input object and adjust it.
129       * The implementation defines the logic of the adjustment and is responsible for
130       * documenting that logic. It may use any method on {@code Temporal} to
131       * query the temporal object and perform the adjustment.
132       * The returned object must have the same observable type as the input object
133       * <p>
134       * The input object must not be altered.
135       * Instead, an adjusted copy of the original must be returned.
136       * This provides equivalent, safe behavior for immutable and mutable temporal objects.
137       * <p>

```

```

138     * The input temporal object may be in a calendar system other than ISO.
139     * Implementations may choose to document compatibility with other calendar systems,
140     * or reject non-ISO temporal objects by {@link TemporalQueries#chronology()} querying the
        chronology}.
141     * <p>
142     * This method may be called from multiple threads in parallel.
143     * It must be thread-safe when invoked.
144     *
145     * @param temporal the temporal object to adjust, not null
146     * @return an object of the same observable type with the adjustment made, not null
147     * @throws DateTimeException if unable to make the adjustment
148     * @throws ArithmeticException if numeric overflow occurs
149     */
150     Temporal adjustInto(Temporal temporal);
151
152 }

```

7.8 TemporalAdjusters.java

Secuencia 119: java/time/temporal/TemporalAdjusters.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2012-2013, Stephen Colebourne & Michael Nascimento Santos
33 *

```

```
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.temporal;
63
64  import static java.time.temporal.ChronoField.DAY_OF_MONTH;
65  import static java.time.temporal.ChronoField.DAY_OF_WEEK;
66  import static java.time.temporal.ChronoField.DAY_OF_YEAR;
67  import static java.time.temporal.ChronoUnit.DAYS;
68  import static java.time.temporal.ChronoUnit.MONTHS;
69  import static java.time.temporal.ChronoUnit.YEARS;
70
71  import java.time.DayOfWeek;
72  import java.time.LocalDate;
73  import java.util.Objects;
74  import java.util.function.UnaryOperator;
75
76  /**
77   * Common and useful TemporalAdjusters.
78   * <p>
79   * Adjusters are a key tool for modifying temporal objects.
80   * They exist to externalize the process of adjustment, permitting different
81   * approaches, as per the strategy design pattern.
82   * Examples might be an adjuster that sets the date avoiding weekends, or one that
83   * sets the date to the last day of the month.
84   * <p>
85   * There are two equivalent ways of using a {@code TemporalAdjuster}.
86   * The first is to invoke the method on the interface directly.
```

```

87  * The second is to use {@link Temporal#with(TemporalAdjuster)}:
88  * <pre>
89  *   // these two lines are equivalent, but the second approach is recommended
90  *   temporal = thisAdjuster.adjustInto(temporal);
91  *   temporal = temporal.with(thisAdjuster);
92  * </pre>
93  * It is recommended to use the second approach, {@code with(TemporalAdjuster)},
94  * as it is a lot clearer to read in code.
95  * <p>
96  * This class contains a standard set of adjusters, available as static methods.
97  * These include:
98  * <ul>
99  * <li>finding the first or last day of the month
100 * <li>finding the first day of next month
101 * <li>finding the first or last day of the year
102 * <li>finding the first day of next year
103 * <li>finding the first or last day-of-week within a month, such as "first Wednesday in June"
104 * <li>finding the next or previous day-of-week, such as "next Thursday"
105 * </ul>
106 *
107 * @implSpec
108 * All the implementations supplied by the static methods are immutable.
109 *
110 * @see TemporalAdjuster
111 * @since 1.8
112 */
113 public final class TemporalAdjusters {
114
115     /**
116      * Private constructor since this is a utility class.
117      */
118     private TemporalAdjusters() {
119     }
120
121     //-----
122     /**
123      * Obtains a {@code TemporalAdjuster} that wraps a date adjuster.
124      * <p>
125      * The {@code TemporalAdjuster} is based on the low level {@code Temporal} interface.
126      * This method allows an adjustment from {@code LocalDate} to {@code LocalDate}
127      * to be wrapped to match the temporal-based interface.
128      * This is provided for convenience to make user-written adjusters simpler.
129      * <p>
130      * In general, user-written adjusters should be static constants:
131      * <pre>{@code
132      *   static TemporalAdjuster TWO_DAYS_LATER =
133      *       TemporalAdjusters.ofDateAdjuster(date -> date.plusDays(2));
134      * }</pre>
135      *
136      * @param dateBasedAdjuster the date-based adjuster, not null
137      * @return the temporal adjuster wrapping on the date adjuster, not null
138      */
139     public static TemporalAdjuster ofDateAdjuster(UnaryOperator<LocalDate> dateBasedAdjuster) {

```



```

140     Objects.requireNonNull(dateBasedAdjuster, "dateBasedAdjuster");
141     return (temporal) -> {
142         LocalDate input = LocalDate.from(temporal);
143         LocalDate output = dateBasedAdjuster.apply(input);
144         return temporal.with(output);
145     };
146 }
147
148 //-----
149 /**
150  * Returns the "first day of month" adjuster, which returns a new date set to
151  * the first day of the current month.
152  * <p>
153  * The ISO calendar system behaves as follows:<br>
154  * The input 2011-01-15 will return 2011-01-01.<br>
155  * The input 2011-02-15 will return 2011-02-01.
156  * <p>
157  * The behavior is suitable for use with most calendar systems.
158  * It is equivalent to:
159  * <pre>
160  *     temporal.with(DAY_OF_MONTH, 1);
161  * </pre>
162  *
163  * @return the first day-of-month adjuster, not null
164  */
165 public static TemporalAdjuster firstDayOfMonth() {
166     return (temporal) -> temporal.with(DAY_OF_MONTH, 1);
167 }
168
169 /**
170  * Returns the "last day of month" adjuster, which returns a new date set to
171  * the last day of the current month.
172  * <p>
173  * The ISO calendar system behaves as follows:<br>
174  * The input 2011-01-15 will return 2011-01-31.<br>
175  * The input 2011-02-15 will return 2011-02-28.<br>
176  * The input 2012-02-15 will return 2012-02-29 (leap year).<br>
177  * The input 2011-04-15 will return 2011-04-30.
178  * <p>
179  * The behavior is suitable for use with most calendar systems.
180  * It is equivalent to:
181  * <pre>
182  *     long lastDay = temporal.range(DAY_OF_MONTH).getMaximum();
183  *     temporal.with(DAY_OF_MONTH, lastDay);
184  * </pre>
185  *
186  * @return the last day-of-month adjuster, not null
187  */
188 public static TemporalAdjuster lastDayOfMonth() {
189     return (temporal) -> temporal.with(DAY_OF_MONTH, temporal.range(DAY_OF_MONTH).getMaximum())
190         ;
191 }

```

```

192  /**
193   * Returns the "first day of next month" adjuster, which returns a new date set to
194   * the first day of the next month.
195   * <p>
196   * The ISO calendar system behaves as follows:<br>
197   * The input 2011-01-15 will return 2011-02-01.<br>
198   * The input 2011-02-15 will return 2011-03-01.
199   * <p>
200   * The behavior is suitable for use with most calendar systems.
201   * It is equivalent to:
202   * <pre>
203   *     temporal.with(DAY_OF_MONTH, 1).plus(1, MONTHS);
204   * </pre>
205   *
206   * @return the first day of next month adjuster, not null
207   */
208  public static TemporalAdjuster firstDayOfNextMonth() {
209      return (temporal) -> temporal.with(DAY_OF_MONTH, 1).plus(1, MONTHS);
210  }
211
212  //-----
213  /**
214   * Returns the "first day of year" adjuster, which returns a new date set to
215   * the first day of the current year.
216   * <p>
217   * The ISO calendar system behaves as follows:<br>
218   * The input 2011-01-15 will return 2011-01-01.<br>
219   * The input 2011-02-15 will return 2011-01-01.<br>
220   * <p>
221   * The behavior is suitable for use with most calendar systems.
222   * It is equivalent to:
223   * <pre>
224   *     temporal.with(DAY_OF_YEAR, 1);
225   * </pre>
226   *
227   * @return the first day-of-year adjuster, not null
228   */
229  public static TemporalAdjuster firstDayOfYear() {
230      return (temporal) -> temporal.with(DAY_OF_YEAR, 1);
231  }
232
233  /**
234   * Returns the "last day of year" adjuster, which returns a new date set to
235   * the last day of the current year.
236   * <p>
237   * The ISO calendar system behaves as follows:<br>
238   * The input 2011-01-15 will return 2011-12-31.<br>
239   * The input 2011-02-15 will return 2011-12-31.<br>
240   * <p>
241   * The behavior is suitable for use with most calendar systems.
242   * It is equivalent to:
243   * <pre>
244   *     long lastDay = temporal.range(DAY_OF_YEAR).getMaximum();

```

```

245     * temporal.with(DAY_OF_YEAR, lastDay);
246     * </pre>
247     *
248     * @return the last day-of-year adjuster, not null
249     */
250     public static TemporalAdjuster lastDayOfYear() {
251         return (temporal) -> temporal.with(DAY_OF_YEAR, temporal.range(DAY_OF_YEAR).getMaximum());
252     }
253
254     /**
255     * Returns the "first day of next year" adjuster, which returns a new date set to
256     * the first day of the next year.
257     * <p>
258     * The ISO calendar system behaves as follows:<br>
259     * The input 2011-01-15 will return 2012-01-01.
260     * <p>
261     * The behavior is suitable for use with most calendar systems.
262     * It is equivalent to:
263     * <pre>
264     *     temporal.with(DAY_OF_YEAR, 1).plus(1, YEARS);
265     * </pre>
266     *
267     * @return the first day of next month adjuster, not null
268     */
269     public static TemporalAdjuster firstDayOfNextYear() {
270         return (temporal) -> temporal.with(DAY_OF_YEAR, 1).plus(1, YEARS);
271     }
272
273     //-----
274     /**
275     * Returns the first in month adjuster, which returns a new date
276     * in the same month with the first matching day-of-week.
277     * This is used for expressions like 'first Tuesday in March'.
278     * <p>
279     * The ISO calendar system behaves as follows:<br>
280     * The input 2011-12-15 for (MONDAY) will return 2011-12-05.<br>
281     * The input 2011-12-15 for (FRIDAY) will return 2011-12-02.<br>
282     * <p>
283     * The behavior is suitable for use with most calendar systems.
284     * It uses the {@code DAY_OF_WEEK} and {@code DAY_OF_MONTH} fields
285     * and the {@code DAYS} unit, and assumes a seven day week.
286     *
287     * @param dayOfWeek the day-of-week, not null
288     * @return the first in month adjuster, not null
289     */
290     public static TemporalAdjuster firstInMonth(DayOfWeek dayOfWeek) {
291         return TemporalAdjusters.dayOfWeekInMonth(1, dayOfWeek);
292     }
293
294     /**
295     * Returns the last in month adjuster, which returns a new date
296     * in the same month with the last matching day-of-week.
297     * This is used for expressions like 'last Tuesday in March'.

```

```

298 * <p>
299 * The ISO calendar system behaves as follows:<br>
300 * The input 2011-12-15 for (MONDAY) will return 2011-12-26.<br>
301 * The input 2011-12-15 for (FRIDAY) will return 2011-12-30.<br>
302 * <p>
303 * The behavior is suitable for use with most calendar systems.
304 * It uses the {@code DAY_OF_WEEK} and {@code DAY_OF_MONTH} fields
305 * and the {@code DAYS} unit, and assumes a seven day week.
306 *
307 * @param dayOfWeek the day-of-week, not null
308 * @return the first in month adjuster, not null
309 */
310 public static TemporalAdjuster lastInMonth(DayOfWeek dayOfWeek) {
311     return TemporalAdjusters.dayOfWeekInMonth(-1, dayOfWeek);
312 }
313
314 /**
315 * Returns the day-of-week in month adjuster, which returns a new date
316 * in the same month with the ordinal day-of-week.
317 * This is used for expressions like the 'second Tuesday in March'.
318 * <p>
319 * The ISO calendar system behaves as follows:<br>
320 * The input 2011-12-15 for (1,TUESDAY) will return 2011-12-06.<br>
321 * The input 2011-12-15 for (2,TUESDAY) will return 2011-12-13.<br>
322 * The input 2011-12-15 for (3,TUESDAY) will return 2011-12-20.<br>
323 * The input 2011-12-15 for (4,TUESDAY) will return 2011-12-27.<br>
324 * The input 2011-12-15 for (5,TUESDAY) will return 2012-01-03.<br>
325 * The input 2011-12-15 for (-1,TUESDAY) will return 2011-12-27 (last in month).<br>
326 * The input 2011-12-15 for (-4,TUESDAY) will return 2011-12-06 (3 weeks before last in month)
327 *   .<br>
328 * The input 2011-12-15 for (-5,TUESDAY) will return 2011-11-29 (4 weeks before last in month)
329 *   .<br>
330 * The input 2011-12-15 for (0,TUESDAY) will return 2011-11-29 (last in previous month).<br>
331 * <p>
332 * For a positive or zero ordinal, the algorithm is equivalent to finding the first
333 * day-of-week that matches within the month and then adding a number of weeks to it.
334 * For a negative ordinal, the algorithm is equivalent to finding the last
335 * day-of-week that matches within the month and then subtracting a number of weeks to it.
336 * The ordinal number of weeks is not validated and is interpreted leniently
337 * according to this algorithm. This definition means that an ordinal of zero finds
338 * the last matching day-of-week in the previous month.
339 * <p>
340 * The behavior is suitable for use with most calendar systems.
341 * It uses the {@code DAY_OF_WEEK} and {@code DAY_OF_MONTH} fields
342 * and the {@code DAYS} unit, and assumes a seven day week.
343 *
344 * @param ordinal the week within the month, unbounded but typically from -5 to 5
345 * @param dayOfWeek the day-of-week, not null
346 * @return the day-of-week in month adjuster, not null
347 */
348 public static TemporalAdjuster dayOfWeekInMonth(int ordinal, DayOfWeek dayOfWeek) {
349     Objects.requireNonNull(dayOfWeek, "dayOfWeek");
350     int dowValue = dayOfWeek.getValue();

```

```

349     if (ordinal >= 0) {
350         return (temporal) -> {
351             Temporal temp = temporal.with(DAY_OF_MONTH, 1);
352             int curDow = temp.get(DAY_OF_WEEK);
353             int dowDiff = (dowValue - curDow + 7) % 7;
354             dowDiff += (ordinal - 1L) * 7L; // safe from overflow
355             return temp.plus(dowDiff, DAYS);
356         };
357     } else {
358         return (temporal) -> {
359             Temporal temp = temporal.with(DAY_OF_MONTH, temporal.range(DAY_OF_MONTH).getMaximum
360                 ());
361             int curDow = temp.get(DAY_OF_WEEK);
362             int daysDiff = dowValue - curDow;
363             daysDiff = (daysDiff == 0 ? 0 : (daysDiff > 0 ? daysDiff - 7 : daysDiff));
364             daysDiff -= (-ordinal - 1L) * 7L; // safe from overflow
365             return temp.plus(daysDiff, DAYS);
366         };
367     }
368 }
369 //-----
370 /**
371  * Returns the next day-of-week adjuster, which adjusts the date to the
372  * first occurrence of the specified day-of-week after the date being adjusted.
373  * <p>
374  * The ISO calendar system behaves as follows:<br>
375  * The input 2011-01-15 (a Saturday) for parameter (MONDAY) will return 2011-01-17 (two days
376  * later).<br>
377  * The input 2011-01-15 (a Saturday) for parameter (WEDNESDAY) will return 2011-01-19 (four
378  * days later).<br>
379  * The input 2011-01-15 (a Saturday) for parameter (SATURDAY) will return 2011-01-22 (seven
380  * days later).
381  * <p>
382  * The behavior is suitable for use with most calendar systems.
383  * It uses the {@code DAY_OF_WEEK} field and the {@code DAYS} unit,
384  * and assumes a seven day week.
385  *
386  * @param dayOfWeek the day-of-week to move the date to, not null
387  * @return the next day-of-week adjuster, not null
388  */
389 public static TemporalAdjuster next(DayOfWeek dayOfWeek) {
390     int dowValue = dayOfWeek.getValue();
391     return (temporal) -> {
392         int calDow = temporal.get(DAY_OF_WEEK);
393         int daysDiff = calDow - dowValue;
394         return temporal.plus(daysDiff >= 0 ? 7 - daysDiff : -daysDiff, DAYS);
395     };
396 }
397 /**
398  * Returns the next-or-same day-of-week adjuster, which adjusts the date to the
399  * first occurrence of the specified day-of-week after the date being adjusted

```

```

398     * unless it is already on that day in which case the same object is returned.
399     * <p>
400     * The ISO calendar system behaves as follows:<br>
401     * The input 2011-01-15 (a Saturday) for parameter (MONDAY) will return 2011-01-17 (two days
      later).<br>
402     * The input 2011-01-15 (a Saturday) for parameter (WEDNESDAY) will return 2011-01-19 (four
      days later).<br>
403     * The input 2011-01-15 (a Saturday) for parameter (SATURDAY) will return 2011-01-15 (same as
      input).
404     * <p>
405     * The behavior is suitable for use with most calendar systems.
406     * It uses the {@code DAY_OF_WEEK} field and the {@code DAYS} unit,
407     * and assumes a seven day week.
408     *
409     * @param dayOfWeek the day-of-week to check for or move the date to, not null
410     * @return the next-or-same day-of-week adjuster, not null
411     */
412     public static TemporalAdjuster nextOrSame(DayOfWeek dayOfWeek) {
413         int dowValue = dayOfWeek.getValue();
414         return (temporal) -> {
415             int calDow = temporal.get(DAY_OF_WEEK);
416             if (calDow == dowValue) {
417                 return temporal;
418             }
419             int daysDiff = calDow - dowValue;
420             return temporal.plus(daysDiff >= 0 ? 7 - daysDiff : -daysDiff, DAYS);
421         };
422     }
423
424     /**
425     * Returns the previous day-of-week adjuster, which adjusts the date to the
426     * first occurrence of the specified day-of-week before the date being adjusted.
427     * <p>
428     * The ISO calendar system behaves as follows:<br>
429     * The input 2011-01-15 (a Saturday) for parameter (MONDAY) will return 2011-01-10 (five days
      earlier).<br>
430     * The input 2011-01-15 (a Saturday) for parameter (WEDNESDAY) will return 2011-01-12 (three
      days earlier).<br>
431     * The input 2011-01-15 (a Saturday) for parameter (SATURDAY) will return 2011-01-08 (seven
      days earlier).
432     * <p>
433     * The behavior is suitable for use with most calendar systems.
434     * It uses the {@code DAY_OF_WEEK} field and the {@code DAYS} unit,
435     * and assumes a seven day week.
436     *
437     * @param dayOfWeek the day-of-week to move the date to, not null
438     * @return the previous day-of-week adjuster, not null
439     */
440     public static TemporalAdjuster previous(DayOfWeek dayOfWeek) {
441         int dowValue = dayOfWeek.getValue();
442         return (temporal) -> {
443             int calDow = temporal.get(DAY_OF_WEEK);
444             int daysDiff = dowValue - calDow;

```

```

445         return temporal.minus(daysDiff >= 0 ? 7 - daysDiff : -daysDiff, DAYS);
446     };
447 }
448
449 /**
450  * Returns the previous-or-same day-of-week adjuster, which adjusts the date to the
451  * first occurrence of the specified day-of-week before the date being adjusted
452  * unless it is already on that day in which case the same object is returned.
453  * <p>
454  * The ISO calendar system behaves as follows:<br>
455  * The input 2011-01-15 (a Saturday) for parameter (MONDAY) will return 2011-01-10 (five days
456  * earlier).<br>
457  * The input 2011-01-15 (a Saturday) for parameter (WEDNESDAY) will return 2011-01-12 (three
458  * days earlier).<br>
459  * The input 2011-01-15 (a Saturday) for parameter (SATURDAY) will return 2011-01-15 (same as
460  * input).
461  * <p>
462  * The behavior is suitable for use with most calendar systems.
463  * It uses the {@code DAY_OF_WEEK} field and the {@code DAYS} unit,
464  * and assumes a seven day week.
465  *
466  * @param dayOfWeek the day-of-week to check for or move the date to, not null
467  * @return the previous-or-same day-of-week adjuster, not null
468  */
469 public static TemporalAdjuster previousOrSame(DayOfWeek dayOfWeek) {
470     int dowValue = dayOfWeek.getValue();
471     return (temporal) -> {
472         int calDow = temporal.get(DAY_OF_WEEK);
473         if (calDow == dowValue) {
474             return temporal;
475         }
476         int daysDiff = dowValue - calDow;
477         return temporal.minus(daysDiff >= 0 ? 7 - daysDiff : -daysDiff, DAYS);
478     };
479 }

```

7.9 TemporalAmount.java

Secuencia 120: java/time/temporal/TemporalAmount.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *

```

```
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26 /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2012, 2013 Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62 package java.time.temporal;
63
64 import java.time.DateTimeException;
65 import java.time.Duration;
```



```

66 import java.time.Period;
67 import java.util.List;
68
69 /**
70  * Framework-level interface defining an amount of time, such as
71  * "6 hours", "8 days" or "2 years and 3 months".
72  * <p>
73  * This is the base interface type for amounts of time.
74  * An amount is distinct from a date or time-of-day in that it is not tied
75  * to any specific point on the time-line.
76  * <p>
77  * The amount can be thought of as a {@code Map} of {@link TemporalUnit} to
78  * {@code long}, exposed via {@link #getUnits()} and {@link #get(TemporalUnit)}.
79  * A simple case might have a single unit-value pair, such as "6 hours".
80  * A more complex case may have multiple unit-value pairs, such as
81  * "7 years, 3 months and 5 days".
82  * <p>
83  * There are two common implementations.
84  * {@link Period} is a date-based implementation, storing years, months and days.
85  * {@link Duration} is a time-based implementation, storing seconds and nanoseconds,
86  * but providing some access using other duration based units such as minutes,
87  * hours and fixed 24-hour days.
88  * <p>
89  * This interface is a framework-level interface that should not be widely
90  * used in application code. Instead, applications should create and pass
91  * around instances of concrete types, such as {@code Period} and {@code Duration}.
92  *
93  * @implSpec
94  * This interface places no restrictions on the mutability of implementations,
95  * however immutability is strongly recommended.
96  *
97  * @since 1.8
98  */
99 public interface TemporalAmount {
100
101     /**
102      * Returns the value of the requested unit.
103      * The units returned from {@link #getUnits()} uniquely define the
104      * value of the {@code TemporalAmount}. A value must be returned
105      * for each unit listed in {@code getUnits}.
106      *
107      * @implSpec
108      * Implementations may declare support for units not listed by {@link #getUnits()}.
109      * Typically, the implementation would define additional units
110      * as conversions for the convenience of developers.
111      *
112      * @param unit the {@code TemporalUnit} for which to return the value
113      * @return the long value of the unit
114      * @throws DateTimeException if a value for the unit cannot be obtained
115      * @throws UnsupportedTemporalTypeException if the {@code unit} is not supported
116      */
117     long get(TemporalUnit unit);
118

```

```

119  /**
120   * Returns the list of units uniquely defining the value of this TemporalAmount.
121   * The list of {@code TemporalUnits} is defined by the implementation class.
122   * The list is a snapshot of the units at the time {@code getUnits}
123   * is called and is not mutable.
124   * The units are ordered from longest duration to the shortest duration
125   * of the unit.
126   *
127   * @implSpec
128   * The list of units completely and uniquely represents the
129   * state of the object without omissions, overlaps or duplication.
130   * The units are in order from longest duration to shortest.
131   *
132   * @return the List of {@code TemporalUnits}; not null
133   */
134  List<TemporalUnit> getUnits();
135
136  /**
137   * Adds to the specified temporal object.
138   * <p>
139   * Adds the amount to the specified temporal object using the logic
140   * encapsulated in the implementing class.
141   * <p>
142   * There are two equivalent ways of using this method.
143   * The first is to invoke this method directly.
144   * The second is to use {@link Temporal#plus(TemporalAmount)}:
145   * <pre>
146   * // These two lines are equivalent, but the second approach is recommended
147   * dateTime = amount.addTo(dateTime);
148   * dateTime = dateTime.plus(adder);
149   * </pre>
150   * It is recommended to use the second approach, {@code plus(TemporalAmount)},
151   * as it is a lot clearer to read in code.
152   *
153   * @implSpec
154   * The implementation must take the input object and add to it.
155   * The implementation defines the logic of the addition and is responsible for
156   * documenting that logic. It may use any method on {@code Temporal} to
157   * query the temporal object and perform the addition.
158   * The returned object must have the same observable type as the input object
159   * <p>
160   * The input object must not be altered.
161   * Instead, an adjusted copy of the original must be returned.
162   * This provides equivalent, safe behavior for immutable and mutable temporal objects.
163   * <p>
164   * The input temporal object may be in a calendar system other than ISO.
165   * Implementations may choose to document compatibility with other calendar systems,
166   * or reject non-ISO temporal objects by {@link TemporalQueries#chronology()} querying the
167   * chronology}.
168   * <p>
169   * This method may be called from multiple threads in parallel.
170   * It must be thread-safe when invoked.
171   *

```

```

171     * @param temporal the temporal object to add the amount to, not null
172     * @return an object of the same observable type with the addition made, not null
173     * @throws DateTimeException if unable to add
174     * @throws ArithmeticException if numeric overflow occurs
175     */
176     Temporal addTo(Temporal temporal);
177
178     /**
179     * Subtracts this object from the specified temporal object.
180     * <p>
181     * Subtracts the amount from the specified temporal object using the logic
182     * encapsulated in the implementing class.
183     * <p>
184     * There are two equivalent ways of using this method.
185     * The first is to invoke this method directly.
186     * The second is to use {@link Temporal#minus(TemporalAmount)}:
187     * <pre>
188     * // these two lines are equivalent, but the second approach is recommended
189     * dateTime = amount.subtractFrom(dateTime);
190     * dateTime = dateTime.minus(amount);
191     * </pre>
192     * It is recommended to use the second approach, {@code minus(TemporalAmount)},
193     * as it is a lot clearer to read in code.
194     *
195     * @implSpec
196     * The implementation must take the input object and subtract from it.
197     * The implementation defines the logic of the subtraction and is responsible for
198     * documenting that logic. It may use any method on {@code Temporal} to
199     * query the temporal object and perform the subtraction.
200     * The returned object must have the same observable type as the input object
201     * <p>
202     * The input object must not be altered.
203     * Instead, an adjusted copy of the original must be returned.
204     * This provides equivalent, safe behavior for immutable and mutable temporal objects.
205     * <p>
206     * The input temporal object may be in a calendar system other than ISO.
207     * Implementations may choose to document compatibility with other calendar systems,
208     * or reject non-ISO temporal objects by {@link TemporalQueries#chronology()} querying the
209     * chronology}.
210     * <p>
211     * This method may be called from multiple threads in parallel.
212     * It must be thread-safe when invoked.
213     *
214     * @param temporal the temporal object to subtract the amount from, not null
215     * @return an object of the same observable type with the subtraction made, not null
216     * @throws DateTimeException if unable to subtract
217     * @throws ArithmeticException if numeric overflow occurs
218     */
219     Temporal subtractFrom(Temporal temporal);

```

Secuencia 121: java/time/temporal/TemporalField.java

```
1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
```

```

53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.temporal;
63
64  import java.time.DateTimeException;
65  import java.time.chrono.Chronology;
66  import java.time.format.ResolverStyle;
67  import java.util.Locale;
68  import java.util.Map;
69  import java.util.Objects;
70
71  /**
72   * A field of date-time, such as month-of-year or hour-of-minute.
73   * <p>
74   * Date and time is expressed using fields which partition the time-line into something
75   * meaningful for humans. Implementations of this interface represent those fields.
76   * <p>
77   * The most commonly used units are defined in {@link ChronoField}.
78   * Further fields are supplied in {@link IsoFields}, {@link WeekFields} and {@link JulianFields}.
79   * Fields can also be written by application code by implementing this interface.
80   * <p>
81   * The field works using double dispatch. Client code calls methods on a date-time like
82   * {@code LocalDateTime} which check if the field is a {@code ChronoField}.
83   * If it is, then the date-time must handle it.
84   * Otherwise, the method call is re-dispatched to the matching method in this interface.
85   *
86   * @implSpec
87   * This interface must be implemented with care to ensure other classes operate correctly.
88   * All implementations that can be instantiated must be final, immutable and thread-safe.
89   * Implementations should be {@code Serializable} where possible.
90   * An enum is as effective implementation choice.
91   *
92   * @since 1.8
93   */
94  public interface TemporalField {
95
96      /**
97       * Gets the display name for the field in the requested locale.
98       * <p>
99       * If there is no display name for the locale then a suitable default must be returned.
100      * <p>
101      * The default implementation must check the locale is not null
102      * and return {@code toString()}.
103      *
104      * @param locale the locale to use, not null
105      * @return the display name for the locale or a suitable default, not null

```

```

106     */
107     default String getDisplayName(Locale locale) {
108         Objects.requireNonNull(locale, "locale");
109         return toString();
110     }
111
112     /**
113      * Gets the unit that the field is measured in.
114      * <p>
115      * The unit of the field is the period that varies within the range.
116      * For example, in the field 'MonthOfYear', the unit is 'Months'.
117      * See also {@link #getRangeUnit()}.
118      *
119      * @return the unit defining the base unit of the field, not null
120      */
121     TemporalUnit getBaseUnit();
122
123     /**
124      * Gets the range that the field is bound by.
125      * <p>
126      * The range of the field is the period that the field varies within.
127      * For example, in the field 'MonthOfYear', the range is 'Years'.
128      * See also {@link #getBaseUnit()}.
129      * <p>
130      * The range is never null. For example, the 'Year' field is shorthand for
131      * 'YearOfForever'. It therefore has a unit of 'Years' and a range of 'Forever'.
132      *
133      * @return the unit defining the range of the field, not null
134      */
135     TemporalUnit getRangeUnit();
136
137     /**
138      * Gets the range of valid values for the field.
139      * <p>
140      * All fields can be expressed as a {@code long} integer.
141      * This method returns an object that describes the valid range for that value.
142      * This method is generally only applicable to the ISO-8601 calendar system.
143      * <p>
144      * Note that the result only describes the minimum and maximum valid values
145      * and it is important not to read too much into them. For example, there
146      * could be values within the range that are invalid for the field.
147      *
148      * @return the range of valid values for the field, not null
149      */
150     ValueRange range();
151
152     //-----
153     /**
154      * Checks if this field represents a component of a date.
155      * <p>
156      * A field is date-based if it can be derived from
157      * {@link ChronoField#EPOCH_DAY EPOCH_DAY}.
158      * Note that it is valid for both {@code isDateBased()} and {@code isTimeBased()}

```

```

159     * to return false, such as when representing a field like minute-of-week.
160     *
161     * @return true if this field is a component of a date
162     */
163     boolean isDateBased();
164
165     /**
166     * Checks if this field represents a component of a time.
167     * <p>
168     * A field is time-based if it can be derived from
169     * {@link ChronoField#NANO_OF_DAY NANO_OF_DAY}.
170     * Note that it is valid for both {@code isDateBased()} and {@code isTimeBased()}
171     * to return false, such as when representing a field like minute-of-week.
172     *
173     * @return true if this field is a component of a time
174     */
175     boolean isTimeBased();
176
177     //-----
178     /**
179     * Checks if this field is supported by the temporal object.
180     * <p>
181     * This determines whether the temporal accessor supports this field.
182     * If this returns false, then the temporal cannot be queried for this field.
183     * <p>
184     * There are two equivalent ways of using this method.
185     * The first is to invoke this method directly.
186     * The second is to use {@link TemporalAccessor#isSupported(TemporalField)}:
187     * <pre>
188     * // these two lines are equivalent, but the second approach is recommended
189     * temporal = thisField.isSupportedBy(temporal);
190     * temporal = temporal.isSupported(thisField);
191     * </pre>
192     * It is recommended to use the second approach, {@code isSupported(TemporalField)},
193     * as it is a lot clearer to read in code.
194     * <p>
195     * Implementations should determine whether they are supported using the fields
196     * available in {@link ChronoField}.
197     *
198     * @param temporal the temporal object to query, not null
199     * @return true if the date-time can be queried for this field, false if not
200     */
201     boolean isSupportedBy(TemporalAccessor temporal);
202
203     /**
204     * Get the range of valid values for this field using the temporal object to
205     * refine the result.
206     * <p>
207     * This uses the temporal object to find the range of valid values for the field.
208     * This is similar to {@link #range()}, however this method refines the result
209     * using the temporal. For example, if the field is {@code DAY_OF_MONTH} the
210     * {@code range} method is not accurate as there are four possible month lengths,
211     * 28, 29, 30 and 31 days. Using this method with a date allows the range to be

```

```

212 * accurate, returning just one of those four options.
213 * <p>
214 * There are two equivalent ways of using this method.
215 * The first is to invoke this method directly.
216 * The second is to use {@link TemporalAccessor#range(TemporalField)}:
217 * <pre>
218 * // these two lines are equivalent, but the second approach is recommended
219 * temporal = thisField.rangeRefinedBy(temporal);
220 * temporal = temporal.range(thisField);
221 * </pre>
222 * It is recommended to use the second approach, {@code range(TemporalField)},
223 * as it is a lot clearer to read in code.
224 * <p>
225 * Implementations should perform any queries or calculations using the fields
226 * available in {@link ChronoField}.
227 * If the field is not supported an {@code UnsupportedOperationException} must be thrown.
228 *
229 * @param temporal the temporal object used to refine the result, not null
230 * @return the range of valid values for this field, not null
231 * @throws DateTimeException if the range for the field cannot be obtained
232 * @throws UnsupportedOperationException if the field is not supported by the temporal
233 */
234 ValueRange rangeRefinedBy(TemporalAccessor temporal);
235
236 /**
237 * Gets the value of this field from the specified temporal object.
238 * <p>
239 * This queries the temporal object for the value of this field.
240 * <p>
241 * There are two equivalent ways of using this method.
242 * The first is to invoke this method directly.
243 * The second is to use {@link TemporalAccessor#getLong(TemporalField)}
244 * (or {@link TemporalAccessor#get(TemporalField)}):
245 * <pre>
246 * // these two lines are equivalent, but the second approach is recommended
247 * temporal = thisField.getFrom(temporal);
248 * temporal = temporal.getLong(thisField);
249 * </pre>
250 * It is recommended to use the second approach, {@code getLong(TemporalField)},
251 * as it is a lot clearer to read in code.
252 * <p>
253 * Implementations should perform any queries or calculations using the fields
254 * available in {@link ChronoField}.
255 * If the field is not supported an {@code UnsupportedOperationException} must be thrown.
256 *
257 * @param temporal the temporal object to query, not null
258 * @return the value of this field, not null
259 * @throws DateTimeException if a value for the field cannot be obtained
260 * @throws UnsupportedOperationException if the field is not supported by the temporal
261 * @throws ArithmeticException if numeric overflow occurs
262 */
263 long getFrom(TemporalAccessor temporal);
264

```



```

265 /**
266  * Returns a copy of the specified temporal object with the value of this field set.
267  * <p>
268  * This returns a new temporal object based on the specified one with the value for
269  * this field changed. For example, on a {@code LocalDate}, this could be used to
270  * set the year, month or day-of-month.
271  * The returned object has the same observable type as the specified object.
272  * <p>
273  * In some cases, changing a field is not fully defined. For example, if the target object is
274  * a date representing the 31st January, then changing the month to February would be unclear.
275  * In cases like this, the implementation is responsible for resolving the result.
276  * Typically it will choose the previous valid date, which would be the last valid
277  * day of February in this example.
278  * <p>
279  * There are two equivalent ways of using this method.
280  * The first is to invoke this method directly.
281  * The second is to use {@link Temporal#with(TemporalField, long)}:
282  * <pre>
283  * // these two lines are equivalent, but the second approach is recommended
284  * temporal = thisField.adjustInto(temporal);
285  * temporal = temporal.with(thisField);
286  * </pre>
287  * It is recommended to use the second approach, {@code with(TemporalField)},
288  * as it is a lot clearer to read in code.
289  * <p>
290  * Implementations should perform any queries or calculations using the fields
291  * available in {@link ChronoField}.
292  * If the field is not supported an {@code UnsupportedOperationException} must be thrown.
293  * <p>
294  * Implementations must not alter the specified temporal object.
295  * Instead, an adjusted copy of the original must be returned.
296  * This provides equivalent, safe behavior for immutable and mutable implementations.
297  *
298  * @param <R> the type of the Temporal object
299  * @param temporal the temporal object to adjust, not null
300  * @param newValue the new value of the field
301  * @return the adjusted temporal object, not null
302  * @throws DateTimeException if the field cannot be set
303  * @throws UnsupportedOperationException if the field is not supported by the temporal
304  * @throws ArithmeticException if numeric overflow occurs
305  */
306 <R extends Temporal> R adjustInto(R temporal, long newValue);
307
308 /**
309  * Resolves this field to provide a simpler alternative or a date.
310  * <p>
311  * This method is invoked during the resolve phase of parsing.
312  * It is designed to allow application defined fields to be simplified into
313  * more standard fields, such as those on {@code ChronoField}, or into a date.
314  * <p>
315  * Applications should not normally invoke this method directly.
316  *
317  * @implSpec

```

```

318 * If an implementation represents a field that can be simplified, or
319 * combined with others, then this method must be implemented.
320 * <p>
321 * The specified map contains the current state of the parse.
322 * The map is mutable and must be mutated to resolve the field and
323 * any related fields. This method will only be invoked during parsing
324 * if the map contains this field, and implementations should therefore
325 * assume this field is present.
326 * <p>
327 * Resolving a field will consist of looking at the value of this field,
328 * and potentially other fields, and either updating the map with a
329 * simpler value, such as a {@code ChronoField}, or returning a
330 * complete {@code ChronoLocalDate}. If a resolve is successful,
331 * the code must remove all the fields that were resolved from the map,
332 * including this field.
333 * <p>
334 * For example, the {@code IsoFields} class contains the quarter-of-year
335 * and day-of-quarter fields. The implementation of this method in that class
336 * resolves the two fields plus the {@link ChronoField#YEAR YEAR} into a
337 * complete {@code LocalDate}. The resolve method will remove all three
338 * fields from the map before returning the {@code LocalDate}.
339 * <p>
340 * A partially complete temporal is used to allow the chronology and zone
341 * to be queried. In general, only the chronology will be needed.
342 * Querying items other than the zone or chronology is undefined and
343 * must not be relied on.
344 * The behavior of other methods such as {@code get}, {@code getLong},
345 * {@code range} and {@code isSupported} is unpredictable and the results undefined.
346 * <p>
347 * If resolution should be possible, but the data is invalid, the resolver
348 * style should be used to determine an appropriate level of leniency, which
349 * may require throwing a {@code DateTimeException} or {@code ArithmeticException}.
350 * If no resolution is possible, the resolve method must return null.
351 * <p>
352 * When resolving time fields, the map will be altered and null returned.
353 * When resolving date fields, the date is normally returned from the method,
354 * with the map altered to remove the resolved fields. However, it would also
355 * be acceptable for the date fields to be resolved into other {@code ChronoField}
356 * instances that can produce a date, such as {@code EPOCH_DAY}.
357 * <p>
358 * Not all {@code TemporalAccessor} implementations are accepted as return values.
359 * Implementations that call this method must accept {@code ChronoLocalDate},
360 * {@code ChronoLocalDateTime}, {@code ChronoZonedDateTime} and {@code LocalTime}.
361 * <p>
362 * The default implementation must return null.
363 *
364 * @param fieldValues the map of fields to values, which can be updated, not null
365 * @param partialTemporal the partially complete temporal to query for zone and
366 * chronology; querying for other things is undefined and not recommended, not null
367 * @param resolverStyle the requested type of resolve, not null
368 * @return the resolved temporal object; null if resolving only
369 * changed the map, or no resolve occurred
370 * @throws ArithmeticException if numeric overflow occurs

```

```

371     * @throws DateTimeException if resolving results in an error. This must not be thrown
372     * by querying a field on the temporal without first checking if it is supported
373     */
374     default TemporalAccessor resolve(
375         Map<TemporalField, Long> fieldValues,
376         TemporalAccessor partialTemporal,
377         ResolverStyle resolverStyle) {
378         return null;
379     }
380
381     /**
382     * Gets a descriptive name for the field.
383     * <p>
384     * The should be of the format 'BaseOfRange', such as 'MonthOfYear',
385     * unless the field has a range of {@code FOREVER}, when only
386     * the base unit is mentioned, such as 'Year' or 'Era'.
387     *
388     * @return the name of the field, not null
389     */
390     @Override
391     String toString();
392
393
394 }

```

7.11 TemporalQueries.java

Secuencia 122: java/time/temporal/TemporalQueries.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25

```

```
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2007-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.temporal;
63
64  import static java.time.temporal.ChronoField.EPOCH_DAY;
65  import static java.time.temporal.ChronoField.NANO_OF_DAY;
66  import static java.time.temporal.ChronoField.OFFSET_SECONDS;
67
68  import java.time.LocalDate;
69  import java.time.LocalDateTime;
70  import java.time.ZoneId;
71  import java.time.ZoneOffset;
72  import java.time.chrono.Chronology;
73
74  /**
75   * Common implementations of {@code TemporalQuery}.
76   * <p>
77   * This class provides common implementations of {@link TemporalQuery}.
78   * These are defined here as they must be constants, and the definition
```

```

79  * of lambdas does not guarantee that. By assigning them once here,
80  * they become 'normal' Java constants.
81  * <p>
82  * Queries are a key tool for extracting information from temporal objects.
83  * They exist to externalize the process of querying, permitting different
84  * approaches, as per the strategy design pattern.
85  * Examples might be a query that checks if the date is the day before February 29th
86  * in a leap year, or calculates the number of days to your next birthday.
87  * <p>
88  * The {@link TemporalField} interface provides another mechanism for querying
89  * temporal objects. That interface is limited to returning a {@code long}.
90  * By contrast, queries can return any type.
91  * <p>
92  * There are two equivalent ways of using a {@code TemporalQuery}.
93  * The first is to invoke the method on this interface directly.
94  * The second is to use {@link TemporalAccessor#query(TemporalQuery)}:
95  * <pre>
96  *     // these two lines are equivalent, but the second approach is recommended
97  *     temporal = thisQuery.queryFrom(temporal);
98  *     temporal = temporal.query(thisQuery);
99  * </pre>
100  * It is recommended to use the second approach, {@code query(TemporalQuery)},
101  * as it is a lot clearer to read in code.
102  * <p>
103  * The most common implementations are method references, such as
104  * {@code LocalDate::from} and {@code ZoneId::from}.
105  * Additional common queries are provided to return:
106  * <ul>
107  * <li> a Chronology,
108  * <li> a LocalDate,
109  * <li> a LocalTime,
110  * <li> a ZoneOffset,
111  * <li> a precision,
112  * <li> a zone, or
113  * <li> a zoneId.
114  * </ul>
115  *
116  * @since 1.8
117  */
118 public final class TemporalQueries {
119     // note that it is vital that each method supplies a constant, not a
120     // calculated value, as they will be checked for using ==
121     // it is also vital that each constant is different (due to the == checking)
122     // as such, alterations to this code must be done with care
123
124     /**
125      * Private constructor since this is a utility class.
126      */
127     private TemporalQueries() {
128     }
129
130     //-----
131     // special constants should be used to extract information from a TemporalAccessor

```

```

132 // that cannot be derived in other ways
133 // Javadoc added here, so as to pretend they are more normal than they really are
134
135 /**
136  * A strict query for the {@code ZoneId}.
137  * <p>
138  * This queries a {@code TemporalAccessor} for the zone.
139  * The zone is only returned if the date-time conceptually contains a {@code ZoneId}.
140  * It will not be returned if the date-time only conceptually has an {@code ZoneOffset}.
141  * Thus a {@link java.time.ZonedDateTime} will return the result of {@code getTime()},
142  * but an {@link java.time.OffsetDateTime} will return null.
143  * <p>
144  * In most cases, applications should use {@link #zone()} as this query is too strict.
145  * <p>
146  * The result from JDK classes implementing {@code TemporalAccessor} is as follows:<br>
147  * {@code LocalDate} returns null<br>
148  * {@code LocalTime} returns null<br>
149  * {@code LocalDateTime} returns null<br>
150  * {@code ZonedDateTime} returns the associated zone<br>
151  * {@code OffsetTime} returns null<br>
152  * {@code OffsetDateTime} returns null<br>
153  * {@code ChronoLocalDate} returns null<br>
154  * {@code ChronoLocalDateTime} returns null<br>
155  * {@code ChronoZonedDateTime} returns the associated zone<br>
156  * {@code Era} returns null<br>
157  * {@code DayOfWeek} returns null<br>
158  * {@code Month} returns null<br>
159  * {@code Year} returns null<br>
160  * {@code YearMonth} returns null<br>
161  * {@code MonthDay} returns null<br>
162  * {@code ZoneOffset} returns null<br>
163  * {@code Instant} returns null<br>
164  *
165  * @return a query that can obtain the zone ID of a temporal, not null
166  */
167 public static TemporalQuery<ZoneId> zoneId() {
168     return TemporalQueries.ZONE_ID;
169 }
170
171 /**
172  * A query for the {@code Chronology}.
173  * <p>
174  * This queries a {@code TemporalAccessor} for the chronology.
175  * If the target {@code TemporalAccessor} represents a date, or part of a date,
176  * then it should return the chronology that the date is expressed in.
177  * As a result of this definition, objects only representing time, such as
178  * {@code LocalTime}, will return null.
179  * <p>
180  * The result from JDK classes implementing {@code TemporalAccessor} is as follows:<br>
181  * {@code LocalDate} returns {@code IsoChronology.INSTANCE}<br>
182  * {@code LocalTime} returns null (does not represent a date)<br>
183  * {@code LocalDateTime} returns {@code IsoChronology.INSTANCE}<br>
184  * {@code ZonedDateTime} returns {@code IsoChronology.INSTANCE}<br>

```

```

185 * {@code OffsetTime} returns null (does not represent a date)<br>
186 * {@code OffsetDateTime} returns {@code IsoChronology.INSTANCE}<br>
187 * {@code ChronoLocalDate} returns the associated chronology<br>
188 * {@code ChronoLocalDateTime} returns the associated chronology<br>
189 * {@code ChronoZonedDateTime} returns the associated chronology<br>
190 * {@code Era} returns the associated chronology<br>
191 * {@code DayOfWeek} returns null (shared across chronologies)<br>
192 * {@code Month} returns {@code IsoChronology.INSTANCE}<br>
193 * {@code Year} returns {@code IsoChronology.INSTANCE}<br>
194 * {@code YearMonth} returns {@code IsoChronology.INSTANCE}<br>
195 * {@code MonthDay} returns null {@code IsoChronology.INSTANCE}<br>
196 * {@code ZoneOffset} returns null (does not represent a date)<br>
197 * {@code Instant} returns null (does not represent a date)<br>
198 * <p>
199 * The method {@link java.time.chrono.Chronology#from(TemporalAccessor)} can be used as a
200 * {@code TemporalQuery} via a method reference, {@code Chronology::from}.
201 * That method is equivalent to this query, except that it throws an
202 * exception if a chronology cannot be obtained.
203 *
204 * @return a query that can obtain the chronology of a temporal, not null
205 */
206 public static TemporalQuery<Chronology> chronology() {
207     return TemporalQueries.CHRONO;
208 }
209
210 /**
211 * A query for the smallest supported unit.
212 * <p>
213 * This queries a {@code TemporalAccessor} for the time precision.
214 * If the target {@code TemporalAccessor} represents a consistent or complete date-time,
215 * date or time then this must return the smallest precision actually supported.
216 * Note that fields such as {@code NANO_OF_DAY} and {@code NANO_OF_SECOND}
217 * are defined to always return ignoring the precision, thus this is the only
218 * way to find the actual smallest supported unit.
219 * For example, were {@code GregorianCalendar} to implement {@code TemporalAccessor}
220 * it would return a precision of {@code MILLIS}.
221 * <p>
222 * The result from JDK classes implementing {@code TemporalAccessor} is as follows:<br>
223 * {@code LocalDate} returns {@code DAYS}<br>
224 * {@code LocalDateTime} returns {@code NANOS}<br>
225 * {@code LocalDateTime} returns {@code NANOS}<br>
226 * {@code ZonedDateTime} returns {@code NANOS}<br>
227 * {@code OffsetTime} returns {@code NANOS}<br>
228 * {@code OffsetDateTime} returns {@code NANOS}<br>
229 * {@code ChronoLocalDate} returns {@code DAYS}<br>
230 * {@code ChronoLocalDateTime} returns {@code NANOS}<br>
231 * {@code ChronoZonedDateTime} returns {@code NANOS}<br>
232 * {@code Era} returns {@code ERAS}<br>
233 * {@code DayOfWeek} returns {@code DAYS}<br>
234 * {@code Month} returns {@code MONTHS}<br>
235 * {@code Year} returns {@code YEARS}<br>
236 * {@code YearMonth} returns {@code MONTHS}<br>
237 * {@code MonthDay} returns null (does not represent a complete date or time)<br>

```



```

238 * {@code ZoneOffset} returns null (does not represent a date or time)<br>
239 * {@code Instant} returns {@code NANOS}<br>
240 *
241 * @return a query that can obtain the precision of a temporal, not null
242 */
243 public static TemporalQuery<TemporalUnit> precision() {
244     return TemporalQueries.PRECISION;
245 }
246
247 //-----
248 // non-special constants are standard queries that derive information from other information
249 /**
250  * A lenient query for the {@code ZoneId}, falling back to the {@code ZoneOffset}.
251  * <p>
252  * This queries a {@code TemporalAccessor} for the zone.
253  * It first tries to obtain the zone, using {@link #zoneId()}.
254  * If that is not found it tries to obtain the {@link #offset()}.
255  * Thus a {@link java.time.ZonedDateTime} will return the result of {@code getZone()},
256  * while an {@link java.time.OffsetDateTime} will return the result of {@code getOffset()}.
257  * <p>
258  * In most cases, applications should use this query rather than {@code #zoneId()}.
259  * <p>
260  * The method {@link ZoneId#from(TemporalAccessor)} can be used as a
261  * {@code TemporalQuery} via a method reference, {@code ZoneId::from}.
262  * That method is equivalent to this query, except that it throws an
263  * exception if a zone cannot be obtained.
264  *
265  * @return a query that can obtain the zone ID or offset of a temporal, not null
266  */
267 public static TemporalQuery<ZoneId> zone() {
268     return TemporalQueries.ZONE;
269 }
270
271 /**
272  * A query for {@code ZoneOffset} returning null if not found.
273  * <p>
274  * This returns a {@code TemporalQuery} that can be used to query a temporal
275  * object for the offset. The query will return null if the temporal
276  * object cannot supply an offset.
277  * <p>
278  * The query implementation examines the {@link ChronoField#OFFSET_SECONDS OFFSET_SECONDS}
279  * field and uses it to create a {@code ZoneOffset}.
280  * <p>
281  * The method {@link java.time.ZoneOffset#from(TemporalAccessor)} can be used as a
282  * {@code TemporalQuery} via a method reference, {@code ZoneOffset::from}.
283  * This query and {@code ZoneOffset::from} will return the same result if the
284  * temporal object contains an offset. If the temporal object does not contain
285  * an offset, then the method reference will throw an exception, whereas this
286  * query will return null.
287  *
288  * @return a query that can obtain the offset of a temporal, not null
289  */
290 public static TemporalQuery<ZoneOffset> offset() {

```



```

291     return TemporalQueries.OFFSET;
292 }
293
294 /**
295  * A query for {@code LocalDate} returning null if not found.
296  * <p>
297  * This returns a {@code TemporalQuery} that can be used to query a temporal
298  * object for the local date. The query will return null if the temporal
299  * object cannot supply a local date.
300  * <p>
301  * The query implementation examines the {@link ChronoField#EPOCH_DAY EPOCH_DAY}
302  * field and uses it to create a {@code LocalDate}.
303  * <p>
304  * The method {@link ZoneOffset#from(TemporalAccessor)} can be used as a
305  * {@code TemporalQuery} via a method reference, {@code LocalDate::from}.
306  * This query and {@code LocalDate::from} will return the same result if the
307  * temporal object contains a date. If the temporal object does not contain
308  * a date, then the method reference will throw an exception, whereas this
309  * query will return null.
310  *
311  * @return a query that can obtain the date of a temporal, not null
312  */
313 public static TemporalQuery<LocalDate> localDate() {
314     return TemporalQueries.LOCAL_DATE;
315 }
316
317 /**
318  * A query for {@code LocalTime} returning null if not found.
319  * <p>
320  * This returns a {@code TemporalQuery} that can be used to query a temporal
321  * object for the local time. The query will return null if the temporal
322  * object cannot supply a local time.
323  * <p>
324  * The query implementation examines the {@link ChronoField#NANO_OF_DAY NANO_OF_DAY}
325  * field and uses it to create a {@code LocalTime}.
326  * <p>
327  * The method {@link ZoneOffset#from(TemporalAccessor)} can be used as a
328  * {@code TemporalQuery} via a method reference, {@code LocalTime::from}.
329  * This query and {@code LocalTime::from} will return the same result if the
330  * temporal object contains a time. If the temporal object does not contain
331  * a time, then the method reference will throw an exception, whereas this
332  * query will return null.
333  *
334  * @return a query that can obtain the time of a temporal, not null
335  */
336 public static TemporalQuery<LocalTime> localTime() {
337     return TemporalQueries.LOCAL_TIME;
338 }
339
340 //-----
341 /**
342  * A strict query for the {@code ZoneId}.
343  */

```

```

344     static final TemporalQuery<ZoneId> ZONE_ID = (temporal) ->
345         temporal.query(TemporalQueries.ZONE_ID);
346
347     /**
348      * A query for the {@code Chronology}.
349      */
350     static final TemporalQuery<Chronology> CHRONO = (temporal) ->
351         temporal.query(TemporalQueries.CHRONO);
352
353     /**
354      * A query for the smallest supported unit.
355      */
356     static final TemporalQuery<TemporalUnit> PRECISION = (temporal) ->
357         temporal.query(TemporalQueries.PRECISION);
358
359     //-----
360     /**
361      * A query for {@code ZoneOffset} returning null if not found.
362      */
363     static final TemporalQuery<ZoneOffset> OFFSET = (temporal) -> {
364         if (temporal.isSupported(OFFSET_SECONDS)) {
365             return ZoneOffset.ofTotalSeconds(temporal.get(OFFSET_SECONDS));
366         }
367         return null;
368     };
369
370     /**
371      * A lenient query for the {@code ZoneId}, falling back to the {@code ZoneOffset}.
372      */
373     static final TemporalQuery<ZoneId> ZONE = (temporal) -> {
374         ZoneId zone = temporal.query(ZONE_ID);
375         return (zone != null ? zone : temporal.query(OFFSET));
376     };
377
378     /**
379      * A query for {@code LocalDate} returning null if not found.
380      */
381     static final TemporalQuery<LocalDate> LOCAL_DATE = (temporal) -> {
382         if (temporal.isSupported(EPOCH_DAY)) {
383             return LocalDate.ofEpochDay(temporal.getLong(EPOCH_DAY));
384         }
385         return null;
386     };
387
388     /**
389      * A query for {@code LocalTime} returning null if not found.
390      */
391     static final TemporalQuery<LocalTime> LOCAL_TIME = (temporal) -> {
392         if (temporal.isSupported(NANO_OF_DAY)) {
393             return LocalTime.ofNanoOfDay(temporal.getLong(NANO_OF_DAY));
394         }
395         return null;
396     };

```

```
397
398 }
```

7.12 TemporalQuery.java

Secuencia 123: java/time/temporal/TemporalQuery.java

```
1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
```

```

48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
62 package java.time.temporal;
63
64 import java.time.DateTimeException;
65
66 /**
67  * Strategy for querying a temporal object.
68  * <p>
69  * Queries are a key tool for extracting information from temporal objects.
70  * They exist to externalize the process of querying, permitting different
71  * approaches, as per the strategy design pattern.
72  * Examples might be a query that checks if the date is the day before February 29th
73  * in a leap year, or calculates the number of days to your next birthday.
74  * <p>
75  * The {@link TemporalField} interface provides another mechanism for querying
76  * temporal objects. That interface is limited to returning a {@code long}.
77  * By contrast, queries can return any type.
78  * <p>
79  * There are two equivalent ways of using a {@code TemporalQuery}.
80  * The first is to invoke the method on this interface directly.
81  * The second is to use {@link TemporalAccessor#query(TemporalQuery)}:
82  * <pre>
83  *     // these two lines are equivalent, but the second approach is recommended
84  *     temporal = thisQuery.queryFrom(temporal);
85  *     temporal = temporal.query(thisQuery);
86  * </pre>
87  * It is recommended to use the second approach, {@code query(TemporalQuery)},
88  * as it is a lot clearer to read in code.
89  * <p>
90  * The most common implementations are method references, such as
91  * {@code LocalDate::from} and {@code ZoneId::from}.
92  * Additional common queries are provided as static methods in {@link TemporalQueries}.
93  *
94  * @implSpec
95  * This interface places no restrictions on the mutability of implementations,
96  * however immutability is strongly recommended.
97  *
98  * @param <R> the type returned from the query
99  *
100 * @since 1.8

```

```

101  */
102  @FunctionalInterface
103  public interface TemporalQuery<R> {
104
105      /**
106       * Queries the specified temporal object.
107       * <p>
108       * This queries the specified temporal object to return an object using the logic
109       * encapsulated in the implementing class.
110       * Examples might be a query that checks if the date is the day before February 29th
111       * in a leap year, or calculates the number of days to your next birthday.
112       * <p>
113       * There are two equivalent ways of using this method.
114       * The first is to invoke this method directly.
115       * The second is to use {@link TemporalAccessor#query(TemporalQuery)}:
116       * <pre>
117       * // these two lines are equivalent, but the second approach is recommended
118       * temporal = thisQuery.queryFrom(temporal);
119       * temporal = temporal.query(thisQuery);
120       * </pre>
121       * It is recommended to use the second approach, {@code query(TemporalQuery)},
122       * as it is a lot clearer to read in code.
123       *
124       * @implSpec
125       * The implementation must take the input object and query it.
126       * The implementation defines the logic of the query and is responsible for
127       * documenting that logic.
128       * It may use any method on {@code TemporalAccessor} to determine the result.
129       * The input object must not be altered.
130       * <p>
131       * The input temporal object may be in a calendar system other than ISO.
132       * Implementations may choose to document compatibility with other calendar systems,
133       * or reject non-ISO temporal objects by {@link TemporalQueries#chronology()} querying the
134       * chronology}.
135       * <p>
136       * This method may be called from multiple threads in parallel.
137       * It must be thread-safe when invoked.
138       *
139       * @param temporal the temporal object to query, not null
140       * @return the queried value, may return null to indicate not found
141       * @throws DateTimeException if unable to query
142       * @throws ArithmeticException if numeric overflow occurs
143       */
144      R queryFrom(TemporalAccessor temporal);
145  }

```

7.13 TemporalUnit.java

Secuencia 124: java/time/temporal/TemporalUnit.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.

```

```
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
```

```

57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.temporal;
63
64  import java.time.DateTimeException;
65  import java.time.Duration;
66  import java.time.LocalDateTime;
67  import java.time.Period;
68  import java.time.chrono.ChronoLocalDate;
69  import java.time.chrono.ChronoLocalDateTime;
70  import java.time.chrono.ChronoZonedDateTime;
71
72  /**
73   * A unit of date-time, such as Days or Hours.
74   * <p>
75   * Measurement of time is built on units, such as years, months, days, hours, minutes and seconds.
76   * Implementations of this interface represent those units.
77   * <p>
78   * An instance of this interface represents the unit itself, rather than an amount of the unit.
79   * See {@link Period} for a class that represents an amount in terms of the common units.
80   * <p>
81   * The most commonly used units are defined in {@link ChronoUnit}.
82   * Further units are supplied in {@link IsoFields}.
83   * Units can also be written by application code by implementing this interface.
84   * <p>
85   * The unit works using double dispatch. Client code calls methods on a date-time like
86   * {@code LocalDateTime} which check if the unit is a {@code ChronoUnit}.
87   * If it is, then the date-time must handle it.
88   * Otherwise, the method call is re-dispatched to the matching method in this interface.
89   *
90   * @implSpec
91   * This interface must be implemented with care to ensure other classes operate correctly.
92   * All implementations that can be instantiated must be final, immutable and thread-safe.
93   * It is recommended to use an enum where possible.
94   *
95   * @since 1.8
96   */
97  public interface TemporalUnit {
98
99      /**
100       * Gets the duration of this unit, which may be an estimate.
101       * <p>
102       * All units return a duration measured in standard nanoseconds from this method.
103       * The duration will be positive and non-zero.
104       * For example, an hour has a duration of {@code 60 * 60 * 1,000,000,000ns}.
105       * <p>
106       * Some units may return an accurate duration while others return an estimate.
107       * For example, days have an estimated duration due to the possibility of
108       * daylight saving time changes.
109       * To determine if the duration is an estimate, use {@link #isDurationEstimated()}.

```

```

110     *
111     * @return the duration of this unit, which may be an estimate, not null
112     */
113     Duration getDuration();
114
115     /**
116     * Checks if the duration of the unit is an estimate.
117     * <p>
118     * All units have a duration, however the duration is not always accurate.
119     * For example, days have an estimated duration due to the possibility of
120     * daylight saving time changes.
121     * This method returns true if the duration is an estimate and false if it is
122     * accurate. Note that accurate/estimated ignores leap seconds.
123     *
124     * @return true if the duration is estimated, false if accurate
125     */
126     boolean isDurationEstimated();
127
128     //-----
129     /**
130     * Checks if this unit represents a component of a date.
131     * <p>
132     * A date is time-based if it can be used to imply meaning from a date.
133     * It must have a {@linkplain #getDuration() duration} that is an integral
134     * multiple of the length of a standard day.
135     * Note that it is valid for both {@code isDateBased()} and {@code isTimeBased()}
136     * to return false, such as when representing a unit like 36 hours.
137     *
138     * @return true if this unit is a component of a date
139     */
140     boolean isDateBased();
141
142     /**
143     * Checks if this unit represents a component of a time.
144     * <p>
145     * A unit is time-based if it can be used to imply meaning from a time.
146     * It must have a {@linkplain #getDuration() duration} that divides into
147     * the length of a standard day without remainder.
148     * Note that it is valid for both {@code isDateBased()} and {@code isTimeBased()}
149     * to return false, such as when representing a unit like 36 hours.
150     *
151     * @return true if this unit is a component of a time
152     */
153     boolean isTimeBased();
154
155     //-----
156     /**
157     * Checks if this unit is supported by the specified temporal object.
158     * <p>
159     * This checks that the implementing date-time can add/subtract this unit.
160     * This can be used to avoid throwing an exception.
161     * <p>
162     * This default implementation derives the value using

```



```

163     * {@link Temporal#plus(long, TemporalUnit)}.
164     *
165     * @param temporal the temporal object to check, not null
166     * @return true if the unit is supported
167     */
168     default boolean isSupportedBy(Temporal temporal) {
169         if (temporal instanceof LocalDateTime) {
170             return isTimeBased();
171         }
172         if (temporal instanceof ChronoLocalDate) {
173             return isDateBased();
174         }
175         if (temporal instanceof ChronoLocalDateTime || temporal instanceof ChronoZonedDateTime) {
176             return true;
177         }
178         try {
179             temporal.plus(1, this);
180             return true;
181         } catch (UnsupportedTemporalTypeException ex) {
182             return false;
183         } catch (RuntimeException ex) {
184             try {
185                 temporal.plus(-1, this);
186                 return true;
187             } catch (RuntimeException ex2) {
188                 return false;
189             }
190         }
191     }
192
193     /**
194     * Returns a copy of the specified temporal object with the specified period added.
195     * <p>
196     * The period added is a multiple of this unit. For example, this method
197     * could be used to add "3 days" to a date by calling this method on the
198     * instance representing "days", passing the date and the period "3".
199     * The period to be added may be negative, which is equivalent to subtraction.
200     * <p>
201     * There are two equivalent ways of using this method.
202     * The first is to invoke this method directly.
203     * The second is to use {@link Temporal#plus(long, TemporalUnit)}:
204     * <pre>
205     * // these two lines are equivalent, but the second approach is recommended
206     * temporal = thisUnit.addTo(temporal);
207     * temporal = temporal.plus(thisUnit);
208     * </pre>
209     * It is recommended to use the second approach, {@code plus(TemporalUnit)},
210     * as it is a lot clearer to read in code.
211     * <p>
212     * Implementations should perform any queries or calculations using the units
213     * available in {@link ChronoUnit} or the fields available in {@link ChronoField}.
214     * If the unit is not supported an {@code UnsupportedTemporalTypeException} must be thrown.
215     * <p>

```

```

216 * Implementations must not alter the specified temporal object.
217 * Instead, an adjusted copy of the original must be returned.
218 * This provides equivalent, safe behavior for immutable and mutable implementations.
219 *
220 * @param <R> the type of the Temporal object
221 * @param temporal the temporal object to adjust, not null
222 * @param amount the amount of this unit to add, positive or negative
223 * @return the adjusted temporal object, not null
224 * @throws DateTimeException if the amount cannot be added
225 * @throws UnsupportedTemporalTypeException if the unit is not supported by the temporal
226 */
227 <R extends Temporal> R addTo(R temporal, long amount);
228
229 //-----
230 /**
231 * Calculates the amount of time between two temporal objects.
232 * <p>
233 * This calculates the amount in terms of this unit. The start and end
234 * points are supplied as temporal objects and must be of compatible types.
235 * The implementation will convert the second type to be an instance of the
236 * first type before the calculating the amount.
237 * The result will be negative if the end is before the start.
238 * For example, the amount in hours between two temporal objects can be
239 * calculated using {@code HOURS.between(startTime, endTime)}.
240 * <p>
241 * The calculation returns a whole number, representing the number of
242 * complete units between the two temporals.
243 * For example, the amount in hours between the times 11:30 and 13:29
244 * will only be one hour as it is one minute short of two hours.
245 * <p>
246 * There are two equivalent ways of using this method.
247 * The first is to invoke this method directly.
248 * The second is to use {@link Temporal#until(Temporal, TemporalUnit)}:
249 * <pre>
250 * // these two lines are equivalent
251 * between = thisUnit.between(start, end);
252 * between = start.until(end, thisUnit);
253 * </pre>
254 * The choice should be made based on which makes the code more readable.
255 * <p>
256 * For example, this method allows the number of days between two dates to
257 * be calculated:
258 * <pre>
259 * long daysBetween = DAYS.between(start, end);
260 * // or alternatively
261 * long daysBetween = start.until(end, DAYS);
262 * </pre>
263 * <p>
264 * Implementations should perform any queries or calculations using the units
265 * available in {@link ChronoUnit} or the fields available in {@link ChronoField}.
266 * If the unit is not supported an {@code UnsupportedTemporalTypeException} must be thrown.
267 * Implementations must not alter the specified temporal objects.
268 *

```

```

269     * @implSpec
270     * Implementations must begin by checking to if the two temporals have the
271     * same type using {@code getClass()}. If they do not, then the result must be
272     * obtained by calling {@code temporal1Inclusive.until(temporal2Exclusive, this)}.
273     *
274     * @param temporal1Inclusive the base temporal object, not null
275     * @param temporal2Exclusive the other temporal object, exclusive, not null
276     * @return the amount of time between temporal1Inclusive and temporal2Exclusive
277     * in terms of this unit; positive if temporal2Exclusive is later than
278     * temporal1Inclusive, negative if earlier
279     * @throws DateTimeException if the amount cannot be calculated, or the end
280     * temporal cannot be converted to the same type as the start temporal
281     * @throws UnsupportedTemporalTypeException if the unit is not supported by the temporal
282     * @throws ArithmeticException if numeric overflow occurs
283     */
284     long between(Temporal temporal1Inclusive, Temporal temporal2Exclusive);
285
286     //-----
287     /**
288     * Gets a descriptive name for the unit.
289     * <p>
290     * This should be in the plural and upper-first camel case, such as 'Days' or 'Minutes'.
291     *
292     * @return the name of this unit, not null
293     */
294     @Override
295     String toString();
296
297 }

```

7.14 UnsupportedTemporalTypeException.java

Secuencia 125: java/time/temporal/UnsupportedTemporalTypeException.java

```

1  /*
2  * Copyright (c) 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *

```

```
21  *
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2013, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.temporal;
63
64  import java.time.DateTimeException;
65
66  /**
67   * UnsupportedTemporalTypeException indicates that a ChronoField or ChronoUnit is
68   * not supported for a Temporal class.
69   *
70   * @implSpec
71   * This class is intended for use in a single thread.
72   *
73   * @since 1.8
```

```

74  */
75  public class UnsupportedTemporalTypeException extends DateTimeException {
76
77      /**
78       * Serialization version.
79       */
80      private static final long serialVersionUID = -6158898438688206006L;
81
82      /**
83       * Constructs a new UnsupportedTemporalTypeException with the specified message.
84       *
85       * @param message the message to use for this exception, may be null
86       */
87      public UnsupportedTemporalTypeException(String message) {
88          super(message);
89      }
90
91      /**
92       * Constructs a new UnsupportedTemporalTypeException with the specified message and cause.
93       *
94       * @param message the message to use for this exception, may be null
95       * @param cause the cause of the exception, may be null
96       */
97      public UnsupportedTemporalTypeException(String message, Throwable cause) {
98          super(message, cause);
99      }
100 }
101 }

```

7.15 ValueRange.java

Secuencia 126: java/time/temporal/ValueRange.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *

```

```
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2011-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.temporal;
63
64  import java.io.IOException;
65  import java.io.InvalidObjectException;
66  import java.io.ObjectInputStream;
67  import java.io.Serializable;
68  import java.time.DateTimeException;
69
70  /**
71   * The range of valid values for a date-time field.
72   * <p>
73   * All {@link TemporalField} instances have a valid range of values.
74   * For example, the ISO day-of-month runs from 1 to somewhere between 28 and 31.
```

```

75  * This class captures that valid range.
76  * <p>
77  * It is important to be aware of the limitations of this class.
78  * Only the minimum and maximum values are provided.
79  * It is possible for there to be invalid values within the outer range.
80  * For example, a weird field may have valid values of 1, 2, 4, 6, 7, thus
81  * have a range of '1 - 7', despite that fact that values 3 and 5 are invalid.
82  * <p>
83  * Instances of this class are not tied to a specific field.
84  *
85  * @implSpec
86  * This class is immutable and thread-safe.
87  *
88  * @since 1.8
89  */
90  public final class ValueRange implements Serializable {
91
92      /**
93       * Serialization version.
94       */
95      private static final long serialVersionUID = -7317881728594519368L;
96
97      /**
98       * The smallest minimum value.
99       */
100     private final long minSmallest;
101     /**
102      * The largest minimum value.
103      */
104     private final long minLargest;
105     /**
106      * The smallest maximum value.
107      */
108     private final long maxSmallest;
109     /**
110      * The largest maximum value.
111      */
112     private final long maxLargest;
113
114     /**
115      * Obtains a fixed value range.
116      * <p>
117      * This factory obtains a range where the minimum and maximum values are fixed.
118      * For example, the ISO month-of-year always runs from 1 to 12.
119      *
120      * @param min the minimum value
121      * @param max the maximum value
122      * @return the ValueRange for min, max, not null
123      * @throws IllegalArgumentException if the minimum is greater than the maximum
124      */
125     public static ValueRange of(long min, long max) {
126         if (min > max) {
127             throw new IllegalArgumentException("Minimum value must be less than maximum value");

```

```
128     }
129     return new ValueRange(min, min, max, max);
130 }
131
132 /**
133  * Obtains a variable value range.
134  * <p>
135  * This factory obtains a range where the minimum value is fixed and the maximum value may vary
136  * .
137  * For example, the ISO day-of-month always starts at 1, but ends between 28 and 31.
138  *
139  * @param min the minimum value
140  * @param maxSmallest the smallest maximum value
141  * @param maxLargest the largest maximum value
142  * @return the ValueRange for min, smallest max, largest max, not null
143  * @throws IllegalArgumentException if
144  *         the minimum is greater than the smallest maximum,
145  *         or the smallest maximum is greater than the largest maximum
146  */
147 public static ValueRange of(long min, long maxSmallest, long maxLargest) {
148     return of(min, min, maxSmallest, maxLargest);
149 }
150
151 /**
152  * Obtains a fully variable value range.
153  * <p>
154  * This factory obtains a range where both the minimum and maximum value may vary.
155  *
156  * @param minSmallest the smallest minimum value
157  * @param minLargest the largest minimum value
158  * @param maxSmallest the smallest maximum value
159  * @param maxLargest the largest maximum value
160  * @return the ValueRange for smallest min, largest min, smallest max, largest max, not null
161  * @throws IllegalArgumentException if
162  *         the smallest minimum is greater than the smallest maximum,
163  *         or the smallest maximum is greater than the largest maximum
164  *         or the largest minimum is greater than the largest maximum
165  */
166 public static ValueRange of(long minSmallest, long minLargest, long maxSmallest, long
167     maxLargest) {
168     if (minSmallest > minLargest) {
169         throw new IllegalArgumentException("Smallest minimum value must be less than largest
170             minimum value");
171     }
172     if (maxSmallest > maxLargest) {
173         throw new IllegalArgumentException("Smallest maximum value must be less than largest
174             maximum value");
175     }
176     if (minLargest > maxLargest) {
177         throw new IllegalArgumentException("Minimum value must be less than maximum value");
178     }
179     return new ValueRange(minSmallest, minLargest, maxSmallest, maxLargest);
180 }
```



```

177
178 /**
179  * Restrictive constructor.
180  *
181  * @param minSmallest the smallest minimum value
182  * @param minLargest  the largest minimum value
183  * @param maxSmallest the smallest minimum value
184  * @param maxLargest  the largest minimum value
185  */
186 private ValueRange(long minSmallest, long minLargest, long maxSmallest, long maxLargest) {
187     this.minSmallest = minSmallest;
188     this.minLargest = minLargest;
189     this.maxSmallest = maxSmallest;
190     this.maxLargest = maxLargest;
191 }
192
193 //-----
194 /**
195  * Is the value range fixed and fully known.
196  * <p>
197  * For example, the ISO day-of-month runs from 1 to between 28 and 31.
198  * Since there is uncertainty about the maximum value, the range is not fixed.
199  * However, for the month of January, the range is always 1 to 31, thus it is fixed.
200  *
201  * @return true if the set of values is fixed
202  */
203 public boolean isFixed() {
204     return minSmallest == minLargest && maxSmallest == maxLargest;
205 }
206
207 //-----
208 /**
209  * Gets the minimum value that the field can take.
210  * <p>
211  * For example, the ISO day-of-month always starts at 1.
212  * The minimum is therefore 1.
213  *
214  * @return the minimum value for this field
215  */
216 public long getMinimum() {
217     return minSmallest;
218 }
219
220 /**
221  * Gets the largest possible minimum value that the field can take.
222  * <p>
223  * For example, the ISO day-of-month always starts at 1.
224  * The largest minimum is therefore 1.
225  *
226  * @return the largest possible minimum value for this field
227  */
228 public long getLargestMinimum() {
229     return minLargest;

```

```

230     }
231
232     /**
233      * Gets the smallest possible maximum value that the field can take.
234      * <p>
235      * For example, the ISO day-of-month runs to between 28 and 31 days.
236      * The smallest maximum is therefore 28.
237      *
238      * @return the smallest possible maximum value for this field
239      */
240     public long getSmallestMaximum() {
241         return maxSmallest;
242     }
243
244     /**
245      * Gets the maximum value that the field can take.
246      * <p>
247      * For example, the ISO day-of-month runs to between 28 and 31 days.
248      * The maximum is therefore 31.
249      *
250      * @return the maximum value for this field
251      */
252     public long getMaximum() {
253         return maxLargest;
254     }
255
256     //-----
257     /**
258      * Checks if all values in the range fit in an {@code int}.
259      * <p>
260      * This checks that all valid values are within the bounds of an {@code int}.
261      * <p>
262      * For example, the ISO month-of-year has values from 1 to 12, which fits in an {@code int}.
263      * By comparison, ISO nano-of-day runs from 1 to 86,400,000,000,000 which does not fit in an {
264      *   {@code int}.
265      * <p>
266      * This implementation uses {@link #getMinimum()} and {@link #getMaximum()}.
267      *
268      * @return true if a valid value always fits in an {@code int}
269      */
270     public boolean isIntValue() {
271         return getMinimum() >= Integer.MIN_VALUE && getMaximum() <= Integer.MAX_VALUE;
272     }
273
274     /**
275      * Checks if the value is within the valid range.
276      * <p>
277      * This checks that the value is within the stored range of values.
278      *
279      * @param value the value to check
280      * @return true if the value is valid
281      */
282     public boolean isValidValue(long value) {

```

```

282     return (value >= getMinimum() && value <= getMaximum());
283 }
284
285 /**
286  * Checks if the value is within the valid range and that all values
287  * in the range fit in an {@code int}.
288  * <p>
289  * This method combines {@link #isIntValue()} and {@link #isValidValue(long)}.
290  *
291  * @param value the value to check
292  * @return true if the value is valid and fits in an {@code int}
293  */
294 public boolean isValidIntValue(long value) {
295     return isIntValue() && isValidValue(value);
296 }
297
298 /**
299  * Checks that the specified value is valid.
300  * <p>
301  * This validates that the value is within the valid range of values.
302  * The field is only used to improve the error message.
303  *
304  * @param value the value to check
305  * @param field the field being checked, may be null
306  * @return the value that was passed in
307  * @see #isValidValue(long)
308  */
309 public long checkValidValue(long value, TemporalField field) {
310     if (isValidValue(value) == false) {
311         throw new DateTimeException(genInvalidFieldMessage(field, value));
312     }
313     return value;
314 }
315
316 /**
317  * Checks that the specified value is valid and fits in an {@code int}.
318  * <p>
319  * This validates that the value is within the valid range of values and that
320  * all valid values are within the bounds of an {@code int}.
321  * The field is only used to improve the error message.
322  *
323  * @param value the value to check
324  * @param field the field being checked, may be null
325  * @return the value that was passed in
326  * @see #isValidIntValue(long)
327  */
328 public int checkValidIntValue(long value, TemporalField field) {
329     if (isValidIntValue(value) == false) {
330         throw new DateTimeException(genInvalidFieldMessage(field, value));
331     }
332     return (int) value;
333 }
334

```

```

335 private String genInvalidFieldMessage(TemporalField field, long value) {
336     if (field != null) {
337         return "Invalid value for " + field + " (valid values " + this + "): " + value;
338     } else {
339         return "Invalid value (valid values " + this + "): " + value;
340     }
341 }
342
343 //-----
344 /**
345  * Restore the state of an ValueRange from the stream.
346  * Check that the values are valid.
347  *
348  * @param s the stream to read
349  * @throws InvalidObjectException if
350  *     the smallest minimum is greater than the smallest maximum,
351  *     or the smallest maximum is greater than the largest maximum
352  *     or the largest minimum is greater than the largest maximum
353  * @throws ClassNotFoundException if a class cannot be resolved
354  */
355 private void readObject(ObjectInputStream s)
356     throws IOException, ClassNotFoundException, InvalidObjectException
357 {
358     s.defaultReadObject();
359     if (minSmallest > minLargest) {
360         throw new InvalidObjectException("Smallest minimum value must be less than largest
361             minimum value");
362     }
363     if (maxSmallest > maxLargest) {
364         throw new InvalidObjectException("Smallest maximum value must be less than largest
365             maximum value");
366     }
367     if (minLargest > maxLargest) {
368         throw new InvalidObjectException("Minimum value must be less than maximum value");
369     }
370 }
371 //-----
372 /**
373  * Checks if this range is equal to another range.
374  * <p>
375  * The comparison is based on the four values, minimum, largest minimum,
376  * smallest maximum and maximum.
377  * Only objects of type {@code ValueRange} are compared, other types return false.
378  *
379  * @param obj the object to check, null returns false
380  * @return true if this is equal to the other range
381  */
382 @Override
383 public boolean equals(Object obj) {
384     if (obj == this) {
385         return true;
386     }

```

```

386     if (obj instanceof ValueRange) {
387         ValueRange other = (ValueRange) obj;
388         return minSmallest == other.minSmallest && minLargest == other.minLargest &&
389             maxSmallest == other.maxSmallest && maxLargest == other.maxLargest;
390     }
391     return false;
392 }
393
394 /**
395  * A hash code for this range.
396  *
397  * @return a suitable hash code
398  */
399 @Override
400 public int hashCode() {
401     long hash = minSmallest + minLargest << 16 + minLargest >> 48 + maxSmallest << 32 +
402         maxSmallest >> 32 + maxLargest << 48 + maxLargest >> 16;
403     return (int) (hash ^ (hash >>> 32));
404 }
405
406 //-----
407 /**
408  * Outputs this range as a {@code String}.
409  * <p>
410  * The format will be '{min}/{largestMin} - {smallestMax}/{max}',
411  * where the largestMin or smallestMax sections may be omitted, together
412  * with associated slash, if they are the same as the min or max.
413  *
414  * @return a string representation of this range, not null
415  */
416 @Override
417 public String toString() {
418     StringBuilder buf = new StringBuilder();
419     buf.append(minSmallest);
420     if (minSmallest != minLargest) {
421         buf.append('/').append(minLargest);
422     }
423     buf.append(" - ").append(maxSmallest);
424     if (maxSmallest != maxLargest) {
425         buf.append('/').append(maxLargest);
426     }
427     return buf.toString();
428 }
429
430 }

```

7.16 WeekFields.java

Secuencia 127: java/time/temporal/WeekFields.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *

```

```
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2011-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
```

```

58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.temporal;
63
64  import static java.time.temporal.ChronoField.DAY_OF_MONTH;
65  import static java.time.temporal.ChronoField.DAY_OF_WEEK;
66  import static java.time.temporal.ChronoField.DAY_OF_YEAR;
67  import static java.time.temporal.ChronoField.MONTH_OF_YEAR;
68  import static java.time.temporal.ChronoField.YEAR;
69  import static java.time.temporal.ChronoUnit.DAYS;
70  import static java.time.temporal.ChronoUnit.FOREVER;
71  import static java.time.temporal.ChronoUnit.MONTHS;
72  import static java.time.temporal.ChronoUnit.WEEKS;
73  import static java.time.temporal.ChronoUnit.YEARS;
74
75  import java.io.IOException;
76  import java.io.InvalidObjectException;
77  import java.io.ObjectInputStream;
78  import java.io.Serializable;
79  import java.time.DateTimeException;
80  import java.time.DayOfWeek;
81  import java.time.chrono.ChronoLocalDate;
82  import java.time.chrono.Chronology;
83  import java.time.format.ResolverStyle;
84  import java.util.Locale;
85  import java.util.Map;
86  import java.util.Objects;
87  import java.util.ResourceBundle;
88  import java.util.concurrent.ConcurrentHashMap;
89  import java.util.concurrent.ConcurrentMap;
90  import sun.util.locale.provider.CalendarDataUtility;
91  import sun.util.locale.provider.LocaleProviderAdapter;
92  import sun.util.locale.provider.LocaleResources;
93
94  /**
95   * Localized definitions of the day-of-week, week-of-month and week-of-year fields.
96   * <p>
97   * A standard week is seven days long, but cultures have different definitions for some
98   * other aspects of a week. This class represents the definition of the week, for the
99   * purpose of providing {@link TemporalField} instances.
100  * <p>
101  * WeekFields provides five fields,
102  * {@link #dayOfWeek()}, {@link #weekOfMonth()}, {@link #weekOfYear()},
103  * {@link #weekOfWeekBasedYear()}, and {@link #weekBasedYear()}
104  * that provide access to the values from any {@linkplain Temporal temporal object}.
105  * <p>
106  * The computations for day-of-week, week-of-month, and week-of-year are based
107  * on the {@linkplain ChronoField#YEAR proleptic-year},
108  * {@linkplain ChronoField#MONTH_OF_YEAR month-of-year},
109  * {@linkplain ChronoField#DAY_OF_MONTH day-of-month}, and
110  * {@linkplain ChronoField#DAY_OF_WEEK ISO day-of-week} which are based on the

```

```

111 * {@linkplain ChronoField#EPOCH_DAY epoch-day} and the chronology.
112 * The values may not be aligned with the {@linkplain ChronoField#YEAR_OF_ERA year-of-Era}
113 * depending on the Chronology.
114 * <p>A week is defined by:
115 * <ul>
116 * <li>The first day-of-week.
117 * For example, the ISO-8601 standard considers Monday to be the first day-of-week.
118 * <li>The minimal number of days in the first week.
119 * For example, the ISO-8601 standard counts the first week as needing at least 4 days.
120 * </ul>
121 * Together these two values allow a year or month to be divided into weeks.
122 *
123 * <h3>Week of Month</h3>
124 * One field is used: week-of-month.
125 * The calculation ensures that weeks never overlap a month boundary.
126 * The month is divided into periods where each period starts on the defined first day-of-week.
127 * The earliest period is referred to as week 0 if it has less than the minimal number of days
128 * and week 1 if it has at least the minimal number of days.
129 *
130 * <table cellpadding="0" cellspacing="3" border="0" style="text-align: left; width: 50%;">
131 * <caption>Examples of WeekFields</caption>
132 * <tr><th>Date</th><td>Day-of-week</td>
133 * <td>First day: Monday<br>Minimal days: 4</td><td>First day: Monday<br>Minimal days: 5</td></tr>
134 * <tr><th>2008-12-31</th><td>Wednesday</td>
135 * <td>Week 5 of December 2008</td><td>Week 5 of December 2008</td></tr>
136 * <tr><th>2009-01-01</th><td>Thursday</td>
137 * <td>Week 1 of January 2009</td><td>Week 0 of January 2009</td></tr>
138 * <tr><th>2009-01-04</th><td>Sunday</td>
139 * <td>Week 1 of January 2009</td><td>Week 0 of January 2009</td></tr>
140 * <tr><th>2009-01-05</th><td>Monday</td>
141 * <td>Week 2 of January 2009</td><td>Week 1 of January 2009</td></tr>
142 * </table>
143 *
144 * <h3>Week of Year</h3>
145 * One field is used: week-of-year.
146 * The calculation ensures that weeks never overlap a year boundary.
147 * The year is divided into periods where each period starts on the defined first day-of-week.
148 * The earliest period is referred to as week 0 if it has less than the minimal number of days
149 * and week 1 if it has at least the minimal number of days.
150 *
151 * <h3>Week Based Year</h3>
152 * Two fields are used for week-based-year, one for the
153 * {@link #weekOfWeekBasedYear() week-of-week-based-year} and one for
154 * {@link #weekBasedYear() week-based-year}. In a week-based-year, each week
155 * belongs to only a single year. Week 1 of a year is the first week that
156 * starts on the first day-of-week and has at least the minimum number of days.
157 * The first and last weeks of a year may contain days from the
158 * previous calendar year or next calendar year respectively.
159 *
160 * <table cellpadding="0" cellspacing="3" border="0" style="text-align: left; width: 50%;">
161 * <caption>Examples of WeekFields for week-based-year</caption>
162 * <tr><th>Date</th><td>Day-of-week</td>
163 * <td>First day: Monday<br>Minimal days: 4</td><td>First day: Monday<br>Minimal days: 5</td></tr>

```



```

164 * <tr><th>2008-12-31</th><td>Wednesday</td>
165 * <td>Week 1 of 2009</td><td>Week 53 of 2008</td></tr>
166 * <tr><th>2009-01-01</th><td>Thursday</td>
167 * <td>Week 1 of 2009</td><td>Week 53 of 2008</td></tr>
168 * <tr><th>2009-01-04</th><td>Sunday</td>
169 * <td>Week 1 of 2009</td><td>Week 53 of 2008</td></tr>
170 * <tr><th>2009-01-05</th><td>Monday</td>
171 * <td>Week 2 of 2009</td><td>Week 1 of 2009</td></tr>
172 * </table>
173 *
174 * @implSpec
175 * This class is immutable and thread-safe.
176 *
177 * @since 1.8
178 */
179 public final class WeekFields implements Serializable {
180     // implementation notes
181     // querying week-of-month or week-of-year should return the week value bound within the month/
182     // year
183     // however, setting the week value should be lenient (use plus/minus weeks)
184     // allow week-of-month outer range [0 to 6]
185     // allow week-of-year outer range [0 to 54]
186     // this is because callers shouldn't be expected to know the details of validity
187     /**
188      * The cache of rules by firstDayOfWeek plus minimalDays.
189      * Initialized first to be available for definition of ISO, etc.
190      */
191     private static final ConcurrentMap<String, WeekFields> CACHE = new ConcurrentHashMap<>(4, 0.75f
192         , 2);
193     /**
194      * The ISO-8601 definition, where a week starts on Monday and the first week
195      * has a minimum of 4 days.
196      * <p>
197      * The ISO-8601 standard defines a calendar system based on weeks.
198      * It uses the week-based-year and week-of-week-based-year concepts to split
199      * up the passage of days instead of the standard year/month/day.
200      * <p>
201      * Note that the first week may start in the previous calendar year.
202      * Note also that the first few days of a calendar year may be in the
203      * week-based-year corresponding to the previous calendar year.
204      */
205     public static final WeekFields ISO = new WeekFields(DayOfWeek.MONDAY, 4);
206
207     /**
208      * The common definition of a week that starts on Sunday and the first week
209      * has a minimum of 1 day.
210      * <p>
211      * Defined as starting on Sunday and with a minimum of 1 day in the month.
212      * This week definition is in use in the US and other European countries.
213      */
214     public static final WeekFields SUNDAY_START = WeekFields.of(DayOfWeek.SUNDAY, 1);

```

```

215
216 /**
217  * The unit that represents week-based-years for the purpose of addition and subtraction.
218  * <p>
219  * This allows a number of week-based-years to be added to, or subtracted from, a date.
220  * The unit is equal to either 52 or 53 weeks.
221  * The estimated duration of a week-based-year is the same as that of a standard ISO
222  * year at {@code 365.2425 Days}.
223  * <p>
224  * The rules for addition add the number of week-based-years to the existing value
225  * for the week-based-year field retaining the week-of-week-based-year
226  * and day-of-week, unless the week number is too large for the target year.
227  * In that case, the week is set to the last week of the year
228  * with the same day-of-week.
229  * <p>
230  * This unit is an immutable and thread-safe singleton.
231  */
232 public static final TemporalUnit WEEK_BASED_YEARS = IsoFields.WEEK_BASED_YEARS;
233
234 /**
235  * Serialization version.
236  */
237 private static final long serialVersionUID = -1177360819670808121L;
238
239 /**
240  * The first day-of-week.
241  */
242 private final DayOfWeek firstDayOfWeek;
243 /**
244  * The minimal number of days in the first week.
245  */
246 private final int minimalDays;
247 /**
248  * The field used to access the computed DayOfWeek.
249  */
250 private final transient TemporalField dayOfWeek = ComputedDayOfField.ofDayOfWeekField(this);
251 /**
252  * The field used to access the computed WeekOfMonth.
253  */
254 private final transient TemporalField weekOfMonth = ComputedDayOfField.ofWeekOfMonthField(this)
    ;
255 /**
256  * The field used to access the computed WeekOfYear.
257  */
258 private final transient TemporalField weekOfYear = ComputedDayOfField.ofWeekOfYearField(this);
259 /**
260  * The field that represents the week-of-week-based-year.
261  * <p>
262  * This field allows the week of the week-based-year value to be queried and set.
263  * <p>
264  * This unit is an immutable and thread-safe singleton.
265  */
266 private final transient TemporalField weekOfWeekBasedYear = ComputedDayOfField.

```

```

    ofWeekOfWeekBasedYearField(this);
267  /**
268   * The field that represents the week-based-year.
269   * <p>
270   * This field allows the week-based-year value to be queried and set.
271   * <p>
272   * This unit is an immutable and thread-safe singleton.
273   */
274  private final transient TemporalField weekBasedYear = ComputedDayOfField.ofWeekBasedYearField(
    this);

275
276  //-----
277  /**
278   * Obtains an instance of {@code WeekFields} appropriate for a locale.
279   * <p>
280   * This will look up appropriate values from the provider of localization data.
281   *
282   * @param locale the locale to use, not null
283   * @return the week-definition, not null
284   */
285  public static WeekFields of(Locale locale) {
286      Objects.requireNonNull(locale, "locale");
287      locale = new Locale(locale.getLanguage(), locale.getCountry()); // eliminate variants
288
289      int calDow = CalendarDataUtility.retrieveFirstDayOfWeek(locale);
290      DayOfWeek dow = DayOfWeek.SUNDAY.plus(calDow - 1);
291      int minDays = CalendarDataUtility.retrieveMinimalDaysInFirstWeek(locale);
292      return WeekFields.of(dow, minDays);
293  }
294
295  /**
296   * Obtains an instance of {@code WeekFields} from the first day-of-week and minimal days.
297   * <p>
298   * The first day-of-week defines the ISO {@code DayOfWeek} that is day 1 of the week.
299   * The minimal number of days in the first week defines how many days must be present
300   * in a month or year, starting from the first day-of-week, before the week is counted
301   * as the first week. A value of 1 will count the first day of the month or year as part
302   * of the first week, whereas a value of 7 will require the whole seven days to be in
303   * the new month or year.
304   * <p>
305   * WeekFields instances are singletons; for each unique combination
306   * of {@code firstDayOfWeek} and {@code minimalDaysInFirstWeek} the
307   * the same instance will be returned.
308   *
309   * @param firstDayOfWeek the first day of the week, not null
310   * @param minimalDaysInFirstWeek the minimal number of days in the first week, from 1 to 7
311   * @return the week-definition, not null
312   * @throws IllegalArgumentException if the minimal days value is less than one
313   *         or greater than 7
314   */
315  public static WeekFields of(DayOfWeek firstDayOfWeek, int minimalDaysInFirstWeek) {
316      String key = firstDayOfWeek.toString() + minimalDaysInFirstWeek;
317      WeekFields rules = CACHE.get(key);

```

```

318     if (rules == null) {
319         rules = new WeekFields(firstDayOfWeek, minimalDaysInFirstWeek);
320         CACHE.putIfAbsent(key, rules);
321         rules = CACHE.get(key);
322     }
323     return rules;
324 }
325
326 //-----
327 /**
328  * Creates an instance of the definition.
329  *
330  * @param firstDayOfWeek the first day of the week, not null
331  * @param minimalDaysInFirstWeek the minimal number of days in the first week, from 1 to 7
332  * @throws IllegalArgumentException if the minimal days value is invalid
333  */
334 private WeekFields(DayOfWeek firstDayOfWeek, int minimalDaysInFirstWeek) {
335     Objects.requireNonNull(firstDayOfWeek, "firstDayOfWeek");
336     if (minimalDaysInFirstWeek < 1 || minimalDaysInFirstWeek > 7) {
337         throw new IllegalArgumentException("Minimal number of days is invalid");
338     }
339     this.firstDayOfWeek = firstDayOfWeek;
340     this.minimalDays = minimalDaysInFirstWeek;
341 }
342
343 //-----
344 /**
345  * Restore the state of a WeekFields from the stream.
346  * Check that the values are valid.
347  *
348  * @param s the stream to read
349  * @throws InvalidObjectException if the serialized object has an invalid
350  *     value for firstDayOfWeek or minimalDays.
351  * @throws ClassNotFoundException if a class cannot be resolved
352  */
353 private void readObject(ObjectInputStream s)
354     throws IOException, ClassNotFoundException, InvalidObjectException
355 {
356     s.defaultReadObject();
357     if (firstDayOfWeek == null) {
358         throw new InvalidObjectException("firstDayOfWeek is null");
359     }
360
361     if (minimalDays < 1 || minimalDays > 7) {
362         throw new InvalidObjectException("Minimal number of days is invalid");
363     }
364 }
365
366 /**
367  * Return the singleton WeekFields associated with the
368  * {@code firstDayOfWeek} and {@code minimalDays}.
369  * @return the singleton WeekFields for the firstDayOfWeek and minimalDays.
370  * @throws InvalidObjectException if the serialized object has invalid

```

```

371     * values for firstDayOfWeek or minimalDays.
372     */
373     private Object readResolve() throws InvalidObjectException {
374         try {
375             return WeekFields.of(firstDayOfWeek, minimalDays);
376         } catch (IllegalArgumentException iae) {
377             throw new InvalidObjectException("Invalid serialized WeekFields: " + iae.getMessage());
378         }
379     }
380
381     //-----
382     /**
383     * Gets the first day-of-week.
384     * <p>
385     * The first day-of-week varies by culture.
386     * For example, the US uses Sunday, while France and the ISO-8601 standard use Monday.
387     * This method returns the first day using the standard {@code DayOfWeek} enum.
388     *
389     * @return the first day-of-week, not null
390     */
391     public DayOfWeek getFirstDayOfWeek() {
392         return firstDayOfWeek;
393     }
394
395     /**
396     * Gets the minimal number of days in the first week.
397     * <p>
398     * The number of days considered to define the first week of a month or year
399     * varies by culture.
400     * For example, the ISO-8601 requires 4 days (more than half a week) to
401     * be present before counting the first week.
402     *
403     * @return the minimal number of days in the first week of a month or year, from 1 to 7
404     */
405     public int getMinimalDaysInFirstWeek() {
406         return minimalDays;
407     }
408
409     //-----
410     /**
411     * Returns a field to access the day of week based on this {@code WeekFields}.
412     * <p>
413     * This is similar to {@link ChronoField#DAY_OF_WEEK} but uses values for
414     * the day-of-week based on this {@code WeekFields}.
415     * The days are numbered from 1 to 7 where the
416     * {@link #getFirstDayOfWeek() first day-of-week} is assigned the value 1.
417     * <p>
418     * For example, if the first day-of-week is Sunday, then that will have the
419     * value 1, with other days ranging from Monday as 2 to Saturday as 7.
420     * <p>
421     * In the resolving phase of parsing, a localized day-of-week will be converted
422     * to a standardized {@code ChronoField} day-of-week.
423     * The day-of-week must be in the valid range 1 to 7.

```

```

424     * Other fields in this class build dates using the standardized day-of-week.
425     *
426     * @return a field providing access to the day-of-week with localized numbering, not null
427     */
428     public TemporalField dayOfWeek() {
429         return dayOfWeek;
430     }
431
432     /**
433     * Returns a field to access the week of month based on this {@code WeekFields}.
434     * <p>
435     * This represents the concept of the count of weeks within the month where weeks
436     * start on a fixed day-of-week, such as Monday.
437     * This field is typically used with {@link WeekFields#dayOfWeek()}.
438     * <p>
439     * Week one (1) is the week starting on the {@link WeekFields#getFirstDayOfWeek}
440     * where there are at least {@link WeekFields#getMinimalDaysInFirstWeek()} days in the month.
441     * Thus, week one may start up to {@code minDays} days before the start of the month.
442     * If the first week starts after the start of the month then the period before is week zero
443     * (0).
444     * <p>
445     * For example:<br>
446     * - if the 1st day of the month is a Monday, week one starts on the 1st and there is no week
447     *   zero<br>
448     * - if the 2nd day of the month is a Monday, week one starts on the 2nd and the 1st is in week
449     *   zero<br>
450     * - if the 4th day of the month is a Monday, week one starts on the 4th and the 1st to 3rd is
451     *   in week zero<br>
452     * - if the 5th day of the month is a Monday, week two starts on the 5th and the 1st to 4th is
453     *   in week one<br>
454     * <p>
455     * This field can be used with any calendar system.
456     * <p>
457     * In the resolving phase of parsing, a date can be created from a year,
458     * week-of-month, month-of-year and day-of-week.
459     * <p>
460     * In {@linkplain ResolverStyle#STRICT strict mode}, all four fields are
461     * validated against their range of valid values. The week-of-month field
462     * is validated to ensure that the resulting month is the month requested.
463     * <p>
464     * In {@linkplain ResolverStyle#SMART smart mode}, all four fields are
465     * validated against their range of valid values. The week-of-month field
466     * is validated from 0 to 6, meaning that the resulting date can be in a
467     * different month to that specified.
468     * <p>
469     * In {@linkplain ResolverStyle#LENIENT lenient mode}, the year and day-of-week
470     * are validated against the range of valid values. The resulting date is calculated
471     * equivalent to the following four stage approach.
472     * First, create a date on the first day of the first week of January in the requested year.
473     * Then take the month-of-year, subtract one, and add the amount in months to the date.
474     * Then take the week-of-month, subtract one, and add the amount in weeks to the date.
475     * Finally, adjust to the correct day-of-week within the localized week.
476     *

```

```

472     * @return a field providing access to the week-of-month, not null
473     */
474     public TemporalField weekOfMonth() {
475         return weekOfMonth;
476     }
477
478     /**
479     * Returns a field to access the week of year based on this {@code WeekFields}.
480     * <p>
481     * This represents the concept of the count of weeks within the year where weeks
482     * start on a fixed day-of-week, such as Monday.
483     * This field is typically used with {@link WeekFields#dayOfWeek()}.
484     * <p>
485     * Week one(1) is the week starting on the {@link WeekFields#getFirstDayOfWeek}
486     * where there are at least {@link WeekFields#getMinimalDaysInFirstWeek()} days in the year.
487     * Thus, week one may start up to {@code minDays} days before the start of the year.
488     * If the first week starts after the start of the year then the period before is week zero (0)
489     * .
490     * <p>
491     * For example:<br>
492     * - if the 1st day of the year is a Monday, week one starts on the 1st and there is no week
493     *   zero<br>
494     * - if the 2nd day of the year is a Monday, week one starts on the 2nd and the 1st is in week
495     *   zero<br>
496     * - if the 4th day of the year is a Monday, week one starts on the 4th and the 1st to 3rd is
497     *   in week zero<br>
498     * - if the 5th day of the year is a Monday, week two starts on the 5th and the 1st to 4th is
499     *   in week one<br>
500     * <p>
501     * This field can be used with any calendar system.
502     * <p>
503     * In the resolving phase of parsing, a date can be created from a year,
504     * week-of-year and day-of-week.
505     * <p>
506     * In {@linkplain ResolverStyle#STRICT strict mode}, all three fields are
507     * validated against their range of valid values. The week-of-year field
508     * is validated to ensure that the resulting year is the year requested.
509     * <p>
510     * In {@linkplain ResolverStyle#SMART smart mode}, all three fields are
511     * validated against their range of valid values. The week-of-year field
512     * is validated from 0 to 54, meaning that the resulting date can be in a
513     * different year to that specified.
514     * <p>
515     * In {@linkplain ResolverStyle#LENIENT lenient mode}, the year and day-of-week
516     * are validated against the range of valid values. The resulting date is calculated
517     * equivalent to the following three stage approach.
518     * First, create a date on the first day of the first week in the requested year.
519     * Then take the week-of-year, subtract one, and add the amount in weeks to the date.
520     * Finally, adjust to the correct day-of-week within the localized week.
521     *
522     * @return a field providing access to the week-of-year, not null
523     */
524     public TemporalField weekOfYear() {

```



```

520         return weekOfYear;
521     }
522
523     /**
524      * Returns a field to access the week of a week-based-year based on this {@code WeekFields}.
525      * <p>
526      * This represents the concept of the count of weeks within the year where weeks
527      * start on a fixed day-of-week, such as Monday and each week belongs to exactly one year.
528      * This field is typically used with {@link WeekFields#dayOfWeek()} and
529      * {@link WeekFields#weekBasedYear()}.
530      * <p>
531      * Week one(1) is the week starting on the {@link WeekFields#getFirstDayOfWeek}
532      * where there are at least {@link WeekFields#getMinimalDaysInFirstWeek()} days in the year.
533      * If the first week starts after the start of the year then the period before
534      * is in the last week of the previous year.
535      * <p>
536      * For example:<br>
537      * - if the 1st day of the year is a Monday, week one starts on the 1st<br>
538      * - if the 2nd day of the year is a Monday, week one starts on the 2nd and
539      *   the 1st is in the last week of the previous year<br>
540      * - if the 4th day of the year is a Monday, week one starts on the 4th and
541      *   the 1st to 3rd is in the last week of the previous year<br>
542      * - if the 5th day of the year is a Monday, week two starts on the 5th and
543      *   the 1st to 4th is in week one<br>
544      * <p>
545      * This field can be used with any calendar system.
546      * <p>
547      * In the resolving phase of parsing, a date can be created from a week-based-year,
548      * week-of-year and day-of-week.
549      * <p>
550      * In {@linkplain ResolverStyle#STRICT strict mode}, all three fields are
551      * validated against their range of valid values. The week-of-year field
552      * is validated to ensure that the resulting week-based-year is the
553      * week-based-year requested.
554      * <p>
555      * In {@linkplain ResolverStyle#SMART smart mode}, all three fields are
556      * validated against their range of valid values. The week-of-week-based-year field
557      * is validated from 1 to 53, meaning that the resulting date can be in the
558      * following week-based-year to that specified.
559      * <p>
560      * In {@linkplain ResolverStyle#LENIENT lenient mode}, the year and day-of-week
561      * are validated against the range of valid values. The resulting date is calculated
562      * equivalent to the following three stage approach.
563      * First, create a date on the first day of the first week in the requested week-based-year.
564      * Then take the week-of-week-based-year, subtract one, and add the amount in weeks to the date
565      * .
566      * Finally, adjust to the correct day-of-week within the localized week.
567      *
568      * @return a field providing access to the week-of-week-based-year, not null
569      */
570     public TemporalField weekOfWeekBasedYear() {
571         return weekOfWeekBasedYear;
572     }

```



```

572
573 /**
574  * Returns a field to access the year of a week-based-year based on this {@code WeekFields}.
575  * <p>
576  * This represents the concept of the year where weeks start on a fixed day-of-week,
577  * such as Monday and each week belongs to exactly one year.
578  * This field is typically used with {@link WeekFields#dayOfWeek()} and
579  * {@link WeekFields#weekOfWeekBasedYear()}.
580  * <p>
581  * Week one(1) is the week starting on the {@link WeekFields#getFirstDayOfWeek}
582  * where there are at least {@link WeekFields#getMinimalDaysInFirstWeek()} days in the year.
583  * Thus, week one may start before the start of the year.
584  * If the first week starts after the start of the year then the period before
585  * is in the last week of the previous year.
586  * <p>
587  * This field can be used with any calendar system.
588  * <p>
589  * In the resolving phase of parsing, a date can be created from a week-based-year,
590  * week-of-year and day-of-week.
591  * <p>
592  * In {@linkplain ResolverStyle#STRICT strict mode}, all three fields are
593  * validated against their range of valid values. The week-of-year field
594  * is validated to ensure that the resulting week-based-year is the
595  * week-based-year requested.
596  * <p>
597  * In {@linkplain ResolverStyle#SMART smart mode}, all three fields are
598  * validated against their range of valid values. The week-of-week-based-year field
599  * is validated from 1 to 53, meaning that the resulting date can be in the
600  * following week-based-year to that specified.
601  * <p>
602  * In {@linkplain ResolverStyle#LENIENT lenient mode}, the year and day-of-week
603  * are validated against the range of valid values. The resulting date is calculated
604  * equivalent to the following three stage approach.
605  * First, create a date on the first day of the first week in the requested week-based-year.
606  * Then take the week-of-week-based-year, subtract one, and add the amount in weeks to the date
607  * .
608  * Finally, adjust to the correct day-of-week within the localized week.
609  *
610  * @return a field providing access to the week-based-year, not null
611  */
612 public TemporalField weekBasedYear() {
613     return weekBasedYear;
614 }
615
616 //-----
617 /**
618  * Checks if this {@code WeekFields} is equal to the specified object.
619  * <p>
620  * The comparison is based on the entire state of the rules, which is
621  * the first day-of-week and minimal days.
622  *
623  * @param object the other rules to compare to, null returns false
624  * @return true if this is equal to the specified rules

```

```

624     */
625     @Override
626     public boolean equals(Object object) {
627         if (this == object) {
628             return true;
629         }
630         if (object instanceof WeekFields) {
631             return hashCode() == object.hashCode();
632         }
633         return false;
634     }
635
636     /**
637      * A hash code for this {@code WeekFields}.
638      *
639      * @return a suitable hash code
640      */
641     @Override
642     public int hashCode() {
643         return firstDayOfWeek.ordinal() * 7 + minimalDays;
644     }
645
646     //-----
647     /**
648      * A string representation of this {@code WeekFields} instance.
649      *
650      * @return the string representation, not null
651      */
652     @Override
653     public String toString() {
654         return "WeekFields[" + firstDayOfWeek + ', ' + minimalDays + ']';
655     }
656
657     //-----
658     /**
659      * Field type that computes DayOfWeek, WeekOfMonth, and WeekOfYear
660      * based on a WeekFields.
661      * A separate Field instance is required for each different WeekFields;
662      * combination of start of week and minimum number of days.
663      * Constructors are provided to create fields for DayOfWeek, WeekOfMonth,
664      * and WeekOfYear.
665      */
666     static class ComputedDayOfField implements TemporalField {
667
668         /**
669          * Returns a field to access the day of week,
670          * computed based on a WeekFields.
671          * <p>
672          * The WeekDefintion of the first day of the week is used with
673          * the ISO DAY_OF_WEEK field to compute week boundaries.
674          */
675         static ComputedDayOfField ofDayOfWeekField(WeekFields weekDef) {
676             return new ComputedDayOfField("DayOfWeek", weekDef, DAYS, WEEKS, DAY_OF_WEEK_RANGE);

```

```

677     }
678
679     /**
680      * Returns a field to access the week of month,
681      * computed based on a WeekFields.
682      * @see WeekFields#weekOfMonth()
683      */
684     static ComputedDayOfField ofWeekOfMonthField(WeekFields weekDef) {
685         return new ComputedDayOfField("WeekOfMonth", weekDef, WEEKS, MONTHS,
686             WEEK_OF_MONTH_RANGE);
687     }
688
689     /**
690      * Returns a field to access the week of year,
691      * computed based on a WeekFields.
692      * @see WeekFields#weekOfYear()
693      */
694     static ComputedDayOfField ofWeekOfYearField(WeekFields weekDef) {
695         return new ComputedDayOfField("WeekOfYear", weekDef, WEEKS, YEARS, WEEK_OF_YEAR_RANGE);
696     }
697
698     /**
699      * Returns a field to access the week of week-based-year,
700      * computed based on a WeekFields.
701      * @see WeekFields#weekOfWeekBasedYear()
702      */
703     static ComputedDayOfField ofWeekOfWeekBasedYearField(WeekFields weekDef) {
704         return new ComputedDayOfField("WeekOfWeekBasedYear", weekDef, WEEKS, IsoFields.
705             WEEK_BASED_YEARS, WEEK_OF_WEEK_BASED_YEAR_RANGE);
706     }
707
708     /**
709      * Returns a field to access the week of week-based-year,
710      * computed based on a WeekFields.
711      * @see WeekFields#weekBasedYear()
712      */
713     static ComputedDayOfField ofWeekBasedYearField(WeekFields weekDef) {
714         return new ComputedDayOfField("WeekBasedYear", weekDef, IsoFields.WEEK_BASED_YEARS,
715             FOREVER, ChronoField.YEAR.range());
716     }
717
718     /**
719      * Return a new week-based-year date of the Chronology, year, week-of-year,
720      * and dow of week.
721      * @param chrono The chronology of the new date
722      * @param yowby the year of the week-based-year
723      * @param wowby the week of the week-based-year
724      * @param dow the day of the week
725      * @return a ChronoLocalDate for the requested year, week of year, and day of week
726      */
727     private ChronoLocalDate ofWeekBasedYear(Chronology chrono,
728         int yowby, int wowby, int dow) {
729         ChronoLocalDate date = chrono.date(yowby, 1, 1);

```

```

727         int ldow = localizedDayOfWeek(date);
728         int offset = startOfWeekOffset(1, ldow);
729
730         // Clamp the week of year to keep it in the same year
731         int yearLen = date.lengthOfYear();
732         int newYearWeek = computeWeek(offset, yearLen + weekDef.getMinimalDaysInFirstWeek());
733         wowby = Math.min(wowby, newYearWeek - 1);
734
735         int days = -offset + (dow - 1) + (wowby - 1) * 7;
736         return date.plus(days, DAYS);
737     }
738
739     private final String name;
740     private final WeekFields weekDef;
741     private final TemporalUnit baseUnit;
742     private final TemporalUnit rangeUnit;
743     private final ValueRange range;
744
745     private ComputedDayOfField(String name, WeekFields weekDef, TemporalUnit baseUnit,
746         TemporalUnit rangeUnit, ValueRange range) {
747         this.name = name;
748         this.weekDef = weekDef;
749         this.baseUnit = baseUnit;
750         this.rangeUnit = rangeUnit;
751         this.range = range;
752     }
753
754     private static final ValueRange DAY_OF_WEEK_RANGE = ValueRange.of(1, 7);
755     private static final ValueRange WEEK_OF_MONTH_RANGE = ValueRange.of(0, 1, 4, 6);
756     private static final ValueRange WEEK_OF_YEAR_RANGE = ValueRange.of(0, 1, 52, 54);
757     private static final ValueRange WEEK_OF_WEEK_BASED_YEAR_RANGE = ValueRange.of(1, 52, 53);
758
759     @Override
760     public long getFrom(TemporalAccessor temporal) {
761         if (rangeUnit == WEEKS) { // day-of-week
762             return localizedDayOfWeek(temporal);
763         } else if (rangeUnit == MONTHS) { // week-of-month
764             return localizedWeekOfMonth(temporal);
765         } else if (rangeUnit == YEARS) { // week-of-year
766             return localizedWeekOfYear(temporal);
767         } else if (rangeUnit == WEEK_BASED_YEARS) {
768             return localizedWeekOfWeekBasedYear(temporal);
769         } else if (rangeUnit == FOREVER) {
770             return localizedWeekBasedYear(temporal);
771         } else {
772             throw new IllegalStateException("unreachable, rangeUnit: " + rangeUnit + ", this: "
773                 + this);
774         }
775     }
776
777     private int localizedDayOfWeek(TemporalAccessor temporal) {
778         int sow = weekDef.getFirstDayOfWeek().getValue();
779         int isoDow = temporal.get(DAY_OF_WEEK);

```

```

778         return Math.floorMod(isoDow - sow, 7) + 1;
779     }
780
781     private int localizedDayOfWeek(int isoDow) {
782         int sow = weekDef.getFirstDayOfWeek().getValue();
783         return Math.floorMod(isoDow - sow, 7) + 1;
784     }
785
786     private long localizedWeekOfMonth(TemporalAccessor temporal) {
787         int dow = localizedDayOfWeek(temporal);
788         int dom = temporal.get(DAY_OF_MONTH);
789         int offset = startOfWeekOffset(dom, dow);
790         return computeWeek(offset, dom);
791     }
792
793     private long localizedWeekOfYear(TemporalAccessor temporal) {
794         int dow = localizedDayOfWeek(temporal);
795         int doy = temporal.get(DAY_OF_YEAR);
796         int offset = startOfWeekOffset(doy, dow);
797         return computeWeek(offset, doy);
798     }
799
800     /**
801      * Returns the year of week-based-year for the temporal.
802      * The year can be the previous year, the current year, or the next year.
803      * @param temporal a date of any chronology, not null
804      * @return the year of week-based-year for the date
805      */
806     private int localizedWeekBasedYear(TemporalAccessor temporal) {
807         int dow = localizedDayOfWeek(temporal);
808         int year = temporal.get(YEAR);
809         int doy = temporal.get(DAY_OF_YEAR);
810         int offset = startOfWeekOffset(doy, dow);
811         int week = computeWeek(offset, doy);
812         if (week == 0) {
813             // Day is in end of week of previous year; return the previous year
814             return year - 1;
815         } else {
816             // If getting close to end of year, use higher precision logic
817             // Check if date of year is in partial week associated with next year
818             ValueRange dayRange = temporal.range(DAY_OF_YEAR);
819             int yearLen = (int)dayRange.getMaximum();
820             int newYearWeek = computeWeek(offset, yearLen + weekDef.getMinimalDaysInFirstWeek()
821             );
822             if (week >= newYearWeek) {
823                 return year + 1;
824             }
825         }
826         return year;
827     }
828
829     /**
830      * Returns the week of week-based-year for the temporal.

```

```

830     * The week can be part of the previous year, the current year,
831     * or the next year depending on the week start and minimum number
832     * of days.
833     * @param temporal a date of any chronology
834     * @return the week of the year
835     * @see #localizedWeekBasedYear(java.time.temporal.TemporalAccessor)
836     */
837     private int localizedWeekOfWeekBasedYear(TemporalAccessor temporal) {
838         int dow = localizedDayOfWeek(temporal);
839         int doy = temporal.get(DAY_OF_YEAR);
840         int offset = startOfWeekOffset(doy, dow);
841         int week = computeWeek(offset, doy);
842         if (week == 0) {
843             // Day is in end of week of previous year
844             // Recompute from the last day of the previous year
845             ChronoLocalDate date = Chronology.from(temporal).date(temporal);
846             date = date.minus(doy, DAYS); // Back down into previous year
847             return localizedWeekOfWeekBasedYear(date);
848         } else if (week > 50) {
849             // If getting close to end of year, use higher precision logic
850             // Check if date of year is in partial week associated with next year
851             ValueRange dayRange = temporal.range(DAY_OF_YEAR);
852             int yearLen = (int) dayRange.getMaximum();
853             int newYearWeek = computeWeek(offset, yearLen + weekDef.getMinimalDaysInFirstWeek()
854             );
855             if (week >= newYearWeek) {
856                 // Overlaps with week of following year; reduce to week in following year
857                 week = week - newYearWeek + 1;
858             }
859             return week;
860         }
861
862         /**
863          * Returns an offset to align week start with a day of month or day of year.
864          *
865          * @param day the day; 1 through infinity
866          * @param dow the day of the week of that day; 1 through 7
867          * @return an offset in days to align a day with the start of the first 'full' week
868          */
869         private int startOfWeekOffset(int day, int dow) {
870             // offset of first day corresponding to the day of week in first 7 days (zero origin)
871             int weekStart = Math.floorMod(day - dow, 7);
872             int offset = -weekStart;
873             if (weekStart + 1 > weekDef.getMinimalDaysInFirstWeek()) {
874                 // The previous week has the minimum days in the current month to be a 'week'
875                 offset = 7 - weekStart;
876             }
877             return offset;
878         }
879
880         /**
881          * Returns the week number computed from the reference day and reference dayOfWeek.

```

```

882      *
883      * @param offset the offset to align a date with the start of week
884      *    from {@link #startOfWeekOffset}.
885      * @param day the day for which to compute the week number
886      * @return the week number where zero is used for a partial week and 1 for the first full
887      *    week
888      */
889      private int computeWeek(int offset, int day) {
890          return ((7 + offset + (day - 1)) / 7);
891      }
892
893      @SuppressWarnings("unchecked")
894      @Override
895      public <R extends Temporal> R adjustInto(R temporal, long newValue) {
896          // Check the new value and get the old value of the field
897          int newVal = range.checkValidIntValue(newValue, this); // lenient check range
898          int currentVal = temporal.get(this);
899          if (newVal == currentVal) {
900              return temporal;
901          }
902
903          if (rangeUnit == FOREVER) { // replace year of WeekBasedYear
904              // Create a new date object with the same chronology,
905              // the desired year and the same week and dow.
906              int idow = temporal.get(weekDef.dayOfWeek);
907              int wowby = temporal.get(weekDef.weekOfWeekBasedYear);
908              return (R) ofWeekBasedYear(Chronology.from(temporal), (int)newValue, wowby, idow);
909          } else {
910              // Compute the difference and add that using the base unit of the field
911              return (R) temporal.plus(newVal - currentVal, baseUnit);
912          }
913      }
914
915      @Override
916      public ChronoLocalDate resolve(
917          Map<TemporalField, Long> fieldValues, TemporalAccessor partialTemporal,
918          ResolverStyle resolverStyle) {
919          final long value = fieldValues.get(this);
920          final int newValue = Math.toIntExact(value); // broad limit makes overflow checking
921              lighter
922          // first convert localized day-of-week to ISO day-of-week
923          // doing this first handles case where both ISO and localized were parsed and might
924              mismatch
925          // day-of-week is always strict as two different day-of-week values makes lenient
926              complex
927          if (rangeUnit == WEEKS) { // day-of-week
928              final int checkedValue = range.checkValidIntValue(value, this); // no leniency as
929                  too complex
930              final int startDow = weekDef.getFirstDayOfWeek().getValue();
931              long isoDow = Math.floorMod((startDow - 1) + (checkedValue - 1), 7) + 1;
932              fieldValues.remove(this);
933              fieldValues.put(DAY_OF_WEEK, isoDow);
934              return null;

```

```

929     }
930
931     // can only build date if ISO day-of-week is present
932     if (fieldValues.containsKey(DAY_OF_WEEK) == false) {
933         return null;
934     }
935     int isoDow = DAY_OF_WEEK.checkValidIntValue(fieldValues.get(DAY_OF_WEEK));
936     int dow = localizedDayOfWeek(isoDow);
937
938     // build date
939     Chronology chrono = Chronology.from(partialTemporal);
940     if (fieldValues.containsKey(YEAR)) {
941         int year = YEAR.checkValidIntValue(fieldValues.get(YEAR)); // validate
942         if (rangeUnit == MONTHS && fieldValues.containsKey(MONTH_OF_YEAR)) { // week-of-
            month
943             long month = fieldValues.get(MONTH_OF_YEAR); // not validated yet
944             return resolveWoM(fieldValues, chrono, year, month, newValue, dow,
                resolverStyle);
945         }
946         if (rangeUnit == YEARS) { // week-of-year
947             return resolveWoY(fieldValues, chrono, year, newValue, dow, resolverStyle);
948         }
949     } else if ((rangeUnit == WEEK_BASED_YEARS || rangeUnit == FOREVER) &&
950         fieldValues.containsKey(weekDef.weekBasedYear) &&
951         fieldValues.containsKey(weekDef.weekOfWeekBasedYear)) { // week-of-week-based-
        year and year-of-week-based-year
952         return resolveWBY(fieldValues, chrono, dow, resolverStyle);
953     }
954     return null;
955 }
956
957 private ChronoLocalDate resolveWoM(
958     Map<TemporalField, Long> fieldValues, Chronology chrono, int year, long month, long
959     wom, int localDow, ResolverStyle resolverStyle) {
960     ChronoLocalDate date;
961     if (resolverStyle == ResolverStyle.LENIENT) {
962         date = chrono.date(year, 1, 1).plus(Math.subtractExact(month, 1), MONTHS);
963         long weeks = Math.subtractExact(wom, localizedWeekOfMonth(date));
964         int days = localDow - localizedDayOfWeek(date); // safe from overflow
965         date = date.plus(Math.addExact(Math.multiplyExact(weeks, 7), days), DAYS);
966     } else {
967         int monthValid = MONTH_OF_YEAR.checkValidIntValue(month); // validate
968         date = chrono.date(year, monthValid, 1);
969         int womInt = range.checkValidIntValue(wom, this); // validate
970         int weeks = (int) (womInt - localizedWeekOfMonth(date)); // safe from overflow
971         int days = localDow - localizedDayOfWeek(date); // safe from overflow
972         date = date.plus(weeks * 7 + days, DAYS);
973         if (resolverStyle == ResolverStyle.STRICT && date.getLong(MONTH_OF_YEAR) != month)
974             {
975                 throw new DateTimeException("Strict mode rejected resolved date as it is in a
                    different month");
976             }
977     }
978 }

```



```

976         fieldValues.remove(this);
977         fieldValues.remove(YEAR);
978         fieldValues.remove(MONTH_OF_YEAR);
979         fieldValues.remove(DAY_OF_WEEK);
980         return date;
981     }
982
983     private ChronoLocalDate resolveWoY(
984         Map<TemporalField, Long> fieldValues, Chronology chrono, int year, long woy, int
          localDow, ResolverStyle resolverStyle) {
985         ChronoLocalDate date = chrono.date(year, 1, 1);
986         if (resolverStyle == ResolverStyle.LENIENT) {
987             long weeks = Math.subtractExact(woy, localizedWeekOfYear(date));
988             int days = localDow - localizedDayOfWeek(date); // safe from overflow
989             date = date.plus(Math.addExact(Math.multiplyExact(weeks, 7), days), DAYS);
990         } else {
991             int womInt = range.checkValidIntValue(woy, this); // validate
992             int weeks = (int) (womInt - localizedWeekOfYear(date)); // safe from overflow
993             int days = localDow - localizedDayOfWeek(date); // safe from overflow
994             date = date.plus(weeks * 7 + days, DAYS);
995             if (resolverStyle == ResolverStyle.STRICT && date.getLong(YEAR) != year) {
996                 throw new DateTimeException("Strict mode rejected resolved date as it is in a
          different year");
997             }
998         }
999         fieldValues.remove(this);
1000        fieldValues.remove(YEAR);
1001        fieldValues.remove(DAY_OF_WEEK);
1002        return date;
1003    }
1004
1005     private ChronoLocalDate resolveWBY(
1006         Map<TemporalField, Long> fieldValues, Chronology chrono, int localDow,
          ResolverStyle resolverStyle) {
1007         int yowby = weekDef.weekBasedYear.range().checkValidIntValue(
          fieldValues.get(weekDef.weekBasedYear), weekDef.weekBasedYear);
1008         ChronoLocalDate date;
1009         if (resolverStyle == ResolverStyle.LENIENT) {
1010             date = ofWeekBasedYear(chrono, yowby, 1, localDow);
1011             long wowby = fieldValues.get(weekDef.weekOfWeekBasedYear);
1012             long weeks = Math.subtractExact(wowby, 1);
1013             date = date.plus(weeks, WEEKS);
1014         } else {
1015             int wowby = weekDef.weekOfWeekBasedYear.range().checkValidIntValue(
          fieldValues.get(weekDef.weekOfWeekBasedYear), weekDef.weekOfWeekBasedYear);
1016             // validate
1017             date = ofWeekBasedYear(chrono, yowby, wowby, localDow);
1018             if (resolverStyle == ResolverStyle.STRICT && localizedWeekBasedYear(date) != yowby)
1019                 {
1020                     throw new DateTimeException("Strict mode rejected resolved date as it is in a
          different week-based-year");
1021                 }
1022         }

```

```
1023         fieldValues.remove(this);
1024         fieldValues.remove(weekDef.weekBasedYear);
1025         fieldValues.remove(weekDef.weekOfWeekBasedYear);
1026         fieldValues.remove(DAY_OF_WEEK);
1027         return date;
1028     }
1029
1030     //-----
1031     @Override
1032     public String getDisplayName(Locale locale) {
1033         Objects.requireNonNull(locale, "locale");
1034         if (rangeUnit == YEARS) { // only have values for week-of-year
1035             LocaleResources lr = LocaleProviderAdapter.getResourceBundleBased()
1036                 .getLocaleResources(locale);
1037             ResourceBundle rb = lr.getJavaTimeFormatData();
1038             return rb.containsKey("field.week") ? rb.getString("field.week") : name;
1039         }
1040         return name;
1041     }
1042
1043     @Override
1044     public TemporalUnit getBaseUnit() {
1045         return baseUnit;
1046     }
1047
1048     @Override
1049     public TemporalUnit getRangeUnit() {
1050         return rangeUnit;
1051     }
1052
1053     @Override
1054     public boolean isDateBased() {
1055         return true;
1056     }
1057
1058     @Override
1059     public boolean isTimeBased() {
1060         return false;
1061     }
1062
1063     @Override
1064     public ValueRange range() {
1065         return range;
1066     }
1067
1068     //-----
1069     @Override
1070     public boolean isSupportedBy(TemporalAccessor temporal) {
1071         if (temporal.isSupported(DAY_OF_WEEK)) {
1072             if (rangeUnit == WEEKS) { // day-of-week
1073                 return true;
1074             } else if (rangeUnit == MONTHS) { // week-of-month
1075                 return temporal.isSupported(DAY_OF_MONTH);
1076             }
1077         }
1078     }
```

```

1076         } else if (rangeUnit == YEARS) { // week-of-year
1077             return temporal.isSupported(DAY_OF_YEAR);
1078         } else if (rangeUnit == WEEK_BASED_YEARS) {
1079             return temporal.isSupported(DAY_OF_YEAR);
1080         } else if (rangeUnit == FOREVER) {
1081             return temporal.isSupported(YEAR);
1082         }
1083     }
1084     return false;
1085 }
1086
1087 @Override
1088 public ValueRange rangeRefinedBy(TemporalAccessor temporal) {
1089     if (rangeUnit == ChronoUnit.WEEKS) { // day-of-week
1090         return range;
1091     } else if (rangeUnit == MONTHS) { // week-of-month
1092         return rangeByWeek(temporal, DAY_OF_MONTH);
1093     } else if (rangeUnit == YEARS) { // week-of-year
1094         return rangeByWeek(temporal, DAY_OF_YEAR);
1095     } else if (rangeUnit == WEEK_BASED_YEARS) {
1096         return rangeWeekOfWeekBasedYear(temporal);
1097     } else if (rangeUnit == FOREVER) {
1098         return YEAR.range();
1099     } else {
1100         throw new IllegalStateException("unreachable, rangeUnit: " + rangeUnit + ", this: "
1101             + this);
1102     }
1103 }
1104
1105 /**
1106  * Map the field range to a week range
1107  * @param temporal the temporal
1108  * @param field the field to get the range of
1109  * @return the ValueRange with the range adjusted to weeks.
1110  */
1111 private ValueRange rangeByWeek(TemporalAccessor temporal, TemporalField field) {
1112     int dow = localizedDayOfWeek(temporal);
1113     int offset = startOfWeekOffset(temporal.get(field), dow);
1114     ValueRange fieldRange = temporal.range(field);
1115     return ValueRange.of(computeWeek(offset, (int) fieldRange.getMinimum()),
1116         computeWeek(offset, (int) fieldRange.getMaximum()));
1117 }
1118
1119 /**
1120  * Map the field range to a week range of a week year.
1121  * @param temporal the temporal
1122  * @return the ValueRange with the range adjusted to weeks.
1123  */
1124 private ValueRange rangeWeekOfWeekBasedYear(TemporalAccessor temporal) {
1125     if (!temporal.isSupported(DAY_OF_YEAR)) {
1126         return WEEK_OF_YEAR_RANGE;
1127     }
1128     int dow = localizedDayOfWeek(temporal);

```

```

1128     int doy = temporal.get(DAY_OF_YEAR);
1129     int offset = startOfWeekOffset(doy, dow);
1130     int week = computeWeek(offset, doy);
1131     if (week == 0) {
1132         // Day is in end of week of previous year
1133         // Recompute from the last day of the previous year
1134         ChronoLocalDate date = Chronology.from(temporal).date(temporal);
1135         date = date.minus(doy + 7, DAYS); // Back down into previous year
1136         return rangeWeekOfWeekBasedYear(date);
1137     }
1138     // Check if day of year is in partial week associated with next year
1139     ValueRange dayRange = temporal.range(DAY_OF_YEAR);
1140     int yearLen = (int)dayRange.getMaximum();
1141     int newYearWeek = computeWeek(offset, yearLen + weekDef.getMinimalDaysInFirstWeek());
1142
1143     if (week >= newYearWeek) {
1144         // Overlaps with weeks of following year; recompute from a week in following year
1145         ChronoLocalDate date = Chronology.from(temporal).date(temporal);
1146         date = date.plus(yearLen - doy + 1 + 7, ChronoUnit.DAYS);
1147         return rangeWeekOfWeekBasedYear(date);
1148     }
1149     return ValueRange.of(1, newYearWeek-1);
1150 }
1151
1152 //-----
1153 @Override
1154 public String toString() {
1155     return name + "[" + weekDef.toString() + "]";
1156 }
1157 }
1158 }

```

7.17 package-info.java

Secuencia 128: java/time/temporal/package-info.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *

```

```
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62
63  /**
64   * <p>
65   * Access to date and time using fields and units, and date time adjusters.
66   * </p>
67   * <p>
68   * This package expands on the base package to provide additional functionality for
69   * more powerful use cases. Support is included for:
70   * </p>
71   * <ul>
```

```

72 * <li>Units of date-time, such as years, months, days and hours</li>
73 * <li>Fields of date-time, such as month-of-year, day-of-week or hour-of-day</li>
74 * <li>Date-time adjustment functions</li>
75 * <li>Different definitions of weeks</li>
76 * </ul>
77 *
78 * <h3>Fields and Units</h3>
79 * <p>
80 * Dates and times are expressed in terms of fields and units.
81 * A unit is used to measure an amount of time, such as years, days or minutes.
82 * All units implement {@link java.time.temporal.TemporalUnit}.
83 * The set of well known units is defined in {@link java.time.temporal.ChronoUnit}, such as {@code
    DAYS}.
84 * The unit interface is designed to allow application defined units.
85 * </p>
86 * <p>
87 * A field is used to express part of a larger date-time, such as year, month-of-year or second-of-
    minute.
88 * All fields implement {@link java.time.temporal.TemporalField}.
89 * The set of well known fields are defined in {@link java.time.temporal.ChronoField}, such as {
    @code HOUR_OF_DAY}.
90 * Additional fields are defined by {@link java.time.temporal.JulianFields}, {@link java.time.
    temporal.WeekFields}
91 * and {@link java.time.temporal.IsoFields}.
92 * The field interface is designed to allow application defined fields.
93 * </p>
94 * <p>
95 * This package provides tools that allow the units and fields of date and time to be accessed
96 * in a general way most suited for frameworks.
97 * {@link java.time.temporal.Temporal} provides the abstraction for date time types that support
    fields.
98 * Its methods support getting the value of a field, creating a new date time with the value of
99 * a field modified, and querying for additional information, typically used to extract the offset
    or time-zone.
100 * </p>
101 * <p>
102 * One use of fields in application code is to retrieve fields for which there is no convenience
    method.
103 * For example, getting the day-of-month is common enough that there is a method on {@code
    LocalDate}
104 * called {@code getDayOfMonth()}. However for more unusual fields it is necessary to use the field
    .
105 * For example, {@code date.get(ChronoField.ALIGNED_WEEK_OF_MONTH)}.
106 * The fields also provide access to the range of valid values.
107 * </p>
108 *
109 * <h3>Adjustment and Query</h3>
110 * <p>
111 * A key part of the date-time problem space is adjusting a date to a new, related value,
112 * such as the "last day of the month", or "next Wednesday".
113 * These are modeled as functions that adjust a base date-time.
114 * The functions implement {@link java.time.temporal.TemporalAdjuster} and operate on {@code
    Temporal}.

```

```

115 * A set of common functions are provided in {@link java.time.temporal.TemporalAdjusters}.
116 * For example, to find the first occurrence of a day-of-week after a given date, use
117 * {@link java.time.temporal.TemporalAdjusters#next(DayOfWeek)}, such as
118 * {@code date.with(next(MONDAY))}.
119 * Applications can also define adjusters by implementing {@link java.time.temporal.
    TemporalAdjuster}.
120 * </p>
121 * <p>
122 * The {@link java.time.temporal.TemporalAmount} interface models amounts of relative time.
123 * </p>
124 * <p>
125 * In addition to adjusting a date-time, an interface is provided to enable querying via
126 * {@link java.time.temporal.TemporalQuery}.
127 * The most common implementations of the query interface are method references.
128 * The {@code from(TemporalAccessor)} methods on major classes can all be used, such as
129 * {@code LocalDate::from} or {@code Month::from}.
130 * Further implementations are provided in {@link java.time.temporal.TemporalQueries} as static
    methods.
131 * Applications can also define queries by implementing {@link java.time.temporal.TemporalQuery}.
132 * </p>
133 *
134 * <h3>Weeks</h3>
135 * <p>
136 * Different locales have different definitions of the week.
137 * For example, in Europe the week typically starts on a Monday, while in the US it starts on a
    Sunday.
138 * The {@link java.time.temporal.WeekFields} class models this distinction.
139 * </p>
140 * <p>
141 * The ISO calendar system defines an additional week-based division of years.
142 * This defines a year based on whole Monday to Monday weeks.
143 * This is modeled in {@link java.time.temporal.IsoFields}.
144 * </p>
145 *
146 * <h3>Package specification</h3>
147 * <p>
148 * Unless otherwise noted, passing a null argument to a constructor or method in any class or
    interface
149 * in this package will cause a {@link java.lang.NullPointerException NullPointerException} to be
    thrown.
150 * The Javadoc "@param" definition is used to summarise the null-behavior.
151 * The "@throws {@link java.lang.NullPointerException}" is not explicitly documented in each method
    .
152 * </p>
153 * <p>
154 * All calculations should check for numeric overflow and throw either an {@link java.lang.
    ArithmeticException}
155 * or a {@link java.time.DateTimeException}.
156 * </p>
157 * @since JDK1.8
158 */
159 package java.time.temporal;

```

8 Time Zone (java.time.zone package)

8.1 Ser.java

Secuencia 129: java/time/zone/Ser.java

```
1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2011-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
```



```
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
62 package java.time.zone;
63
64 import java.io.DataInput;
65 import java.io.DataOutput;
66 import java.io.Externalizable;
67 import java.io.IOException;
68 import java.io.InvalidClassException;
69 import java.io.ObjectInput;
70 import java.io.ObjectOutput;
71 import java.io.StreamCorruptedException;
72 import java.time.ZoneOffset;
73
74 /**
75  * The shared serialization delegate for this package.
76  *
77  * @implNote
78  * This class is mutable and should be created once per serialization.
79  *
80  * @serial include
81  * @since 1.8
82  */
83 final class Ser implements Externalizable {
84
85     /**
86      * Serialization version.
87      */
88     private static final long serialVersionUID = -8885321777449118786L;
89
90     /** Type for ZoneRules. */
91     static final byte ZRULES = 1;
92     /** Type for ZoneOffsetTransition. */
93     static final byte ZOT = 2;
94     /** Type for ZoneOffsetTransition. */
95     static final byte ZOTRULE = 3;
96
97     /** The type being serialized. */
98     private byte type;
99     /** The object being serialized. */
100    private Object object;
```

```

101
102  /**
103   * Constructor for deserialization.
104   */
105  public Ser() {
106  }
107
108  /**
109   * Creates an instance for serialization.
110   *
111   * @param type the type
112   * @param object the object
113   */
114  Ser(byte type, Object object) {
115      this.type = type;
116      this.object = object;
117  }
118
119  //-----
120  /**
121   * Implements the {@code Externalizable} interface to write the object.
122   * @serialData
123   * Each serializable class is mapped to a type that is the first byte
124   * in the stream. Refer to each class {@code writeReplace}
125   * serialized form for the value of the type and sequence of values for the type.
126   *
127   * <ul>
128   * <li><a href="../../serialized-form.html#java.time.zone.ZoneRules">ZoneRules.writeReplace
129   * </a>
130   * <li><a href="../../serialized-form.html#java.time.zone.ZoneOffsetTransition">
131   * ZoneOffsetTransition.writeReplace</a>
132   * <li><a href="../../serialized-form.html#java.time.zone.ZoneOffsetTransitionRule">
133   * ZoneOffsetTransitionRule.writeReplace</a>
134   * </ul>
135   *
136   * @param out the data stream to write to, not null
137   */
138  @Override
139  public void writeExternal(ObjectOutput out) throws IOException {
140      writeInternal(type, object, out);
141  }
142
143  static void write(Object object, DataOutput out) throws IOException {
144      writeInternal(ZRULES, object, out);
145  }
146
147  private static void writeInternal(byte type, Object object, DataOutput out) throws IOException
148  {
149      out.writeByte(type);
150      switch (type) {
151          case ZRULES:
152              ((ZoneRules) object).writeExternal(out);
153              break;

```

```

150         case ZOT:
151             ((ZoneOffsetTransition) object).writeExternal(out);
152             break;
153         case ZOTRULE:
154             ((ZoneOffsetTransitionRule) object).writeExternal(out);
155             break;
156         default:
157             throw new InvalidClassException("Unknown serialized type");
158     }
159 }
160
161 //-----
162 /**
163  * Implements the {@code Externalizable} interface to read the object.
164  * @serialData
165  * The streamed type and parameters defined by the type's {@code writeReplace}
166  * method are read and passed to the corresponding static factory for the type
167  * to create a new instance. That instance is returned as the de-serialized
168  * {@code Ser} object.
169  *
170  * <ul>
171  * <li><a href="../../serialized-form.html#java.time.zone.ZoneRules">ZoneRules</a>
172  * - {@code ZoneRules.of(standardTransitions, standardOffsets, savingsInstantTransitions,
173   *   wallOffsets, lastRules);}
174  * <li><a href="../../serialized-form.html#java.time.zone.ZoneOffsetTransition">
175   *   ZoneOffsetTransition</a>
176  * - {@code ZoneOffsetTransition.of(LocalDateTime.ofEpochSecond(epochSecond), offsetBefore,
177   *   offsetAfter);}
178  * <li><a href="../../serialized-form.html#java.time.zone.ZoneOffsetTransitionRule">
179   *   ZoneOffsetTransitionRule</a>
180  * - {@code ZoneOffsetTransitionRule.of(month, dom, dow, time, timeEndOfDay, timeDefinition,
181   *   standardOffset, offsetBefore, offsetAfter);}
182  * </ul>
183  * @param in the data to read, not null
184  */
185 @Override
186 public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
187     type = in.readByte();
188     object = readInternal(type, in);
189 }
190
191 static Object read(DataInput in) throws IOException, ClassNotFoundException {
192     byte type = in.readByte();
193     return readInternal(type, in);
194 }
195
196 private static Object readInternal(byte type, DataInput in) throws IOException,
197     ClassNotFoundException {
198     switch (type) {
199         case ZRULES:
200             return ZoneRules.readExternal(in);
201         case ZOT:
202             return ZoneOffsetTransition.readExternal(in);

```

```

197         case ZOTRULE:
198             return ZoneOffsetTransitionRule.readExternal(in);
199         default:
200             throw new StreamCorruptedException("Unknown serialized type");
201     }
202 }
203
204 /**
205  * Returns the object that will replace this one.
206  *
207  * @return the read object, should never be null
208  */
209 private Object readResolve() {
210     return object;
211 }
212
213 //-----
214 /**
215  * Writes the state to the stream.
216  *
217  * @param offset the offset, not null
218  * @param out the output stream, not null
219  * @throws IOException if an error occurs
220  */
221 static void writeOffset(ZoneOffset offset, DataOutput out) throws IOException {
222     final int offsetSecs = offset.getTotalSeconds();
223     int offsetByte = offsetSecs % 900 == 0 ? offsetSecs / 900 : 127; // compress to -72 to +72
224     out.writeByte(offsetByte);
225     if (offsetByte == 127) {
226         out.writeInt(offsetSecs);
227     }
228 }
229
230 /**
231  * Reads the state from the stream.
232  *
233  * @param in the input stream, not null
234  * @return the created object, not null
235  * @throws IOException if an error occurs
236  */
237 static ZoneOffset readOffset(DataInput in) throws IOException {
238     int offsetByte = in.readByte();
239     return (offsetByte == 127 ? ZoneOffset.ofTotalSeconds(in.readInt()) : ZoneOffset.
240         ofTotalSeconds(offsetByte * 900));
241 }
242
243 //-----
244 /**
245  * Writes the state to the stream.
246  *
247  * @param epochSec the epoch seconds, not null
248  * @param out the output stream, not null
249  * @throws IOException if an error occurs

```

```

249  */
250  static void writeEpochSec(long epochSec, DataOutput out) throws IOException {
251      if (epochSec >= -4575744000L && epochSec < 10413792000L && epochSec % 900 == 0) { //
          quarter hours between 1825 and 2300
252          int store = (int) ((epochSec + 4575744000L) / 900);
253          out.writeByte((store >>> 16) & 255);
254          out.writeByte((store >>> 8) & 255);
255          out.writeByte(store & 255);
256      } else {
257          out.writeByte(255);
258          out.writeLong(epochSec);
259      }
260  }
261
262  /**
263   * Reads the state from the stream.
264   *
265   * @param in the input stream, not null
266   * @return the epoch seconds, not null
267   * @throws IOException if an error occurs
268   */
269  static long readEpochSec(DataInput in) throws IOException {
270      int hiByte = in.readByte() & 255;
271      if (hiByte == 255) {
272          return in.readLong();
273      } else {
274          int midByte = in.readByte() & 255;
275          int loByte = in.readByte() & 255;
276          long tot = ((hiByte << 16) + (midByte << 8) + loByte);
277          return (tot * 900) - 4575744000L;
278      }
279  }
280
281  }

```

8.2 TzdbZoneRulesProvider.java

Secuencia 130: java/time/zone/TzdbZoneRulesProvider.java

```

1  /*
2   * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3   * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4   *
5   *
6   *
7   *
8   *
9   *
10  *
11  *
12  *
13  *
14  *
15  *

```

```
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26 /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2009-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62 package java.time.zone;
63
64 import java.io.ByteArrayInputStream;
65 import java.io.BufferedInputStream;
66 import java.io.DataInputStream;
67 import java.io.File;
68 import java.io.FileInputStream;
```

```
69 import java.io.IOException;
70 import java.io.StreamCorruptedException;
71 import java.util.Arrays;
72 import java.util.HashSet;
73 import java.util.List;
74 import java.util.Map;
75 import java.util.NavigableMap;
76 import java.util.Objects;
77 import java.util.Set;
78 import java.util.TreeMap;
79 import java.util.concurrent.ConcurrentHashMap;
80
81 /**
82  * Loads time-zone rules for 'TZDB'.
83  *
84  * @since 1.8
85  */
86 final class TzdbZoneRulesProvider extends ZoneRulesProvider {
87
88     /**
89      * All the regions that are available.
90      */
91     private List<String> regionIds;
92     /**
93      * Version Id of this tzdb rules
94      */
95     private String versionId;
96     /**
97      * Region to rules mapping
98      */
99     private final Map<String, Object> regionToRules = new ConcurrentHashMap<>();
100
101     /**
102      * Creates an instance.
103      * Created by the {@code ServiceLoader}.
104      *
105      * @throws ZoneRulesException if unable to load
106      */
107     public TzdbZoneRulesProvider() {
108         try {
109             String libDir = System.getProperty("java.home") + File.separator + "lib";
110             try (DataInputStream dis = new DataInputStream(
111                 new BufferedInputStream(new FileInputStream(
112                     new File(libDir, "tzdb.dat"))))) {
113                 load(dis);
114             }
115         } catch (Exception ex) {
116             throw new ZoneRulesException("Unable to load TZDB time-zone rules", ex);
117         }
118     }
119
120     @Override
121     protected Set<String> provideZoneIds() {
```

```

122     return new HashSet<>(regionIds);
123 }
124
125 @Override
126 protected ZoneRules provideRules(String zoneId, boolean forCaching) {
127     // forCaching flag is ignored because this is not a dynamic provider
128     Object obj = regionToRules.get(zoneId);
129     if (obj == null) {
130         throw new ZoneRulesException("Unknown time-zone ID: " + zoneId);
131     }
132     try {
133         if (obj instanceof byte[]) {
134             byte[] bytes = (byte[]) obj;
135             DataInputStream dis = new DataInputStream(new ByteArrayInputStream(bytes));
136             obj = Ser.read(dis);
137             regionToRules.put(zoneId, obj);
138         }
139         return (ZoneRules) obj;
140     } catch (Exception ex) {
141         throw new ZoneRulesException("Invalid binary time-zone data: TZDB:" + zoneId + ",
142                                     version: " + versionId, ex);
143     }
144 }
145
146 @Override
147 protected NavigableMap<String, ZoneRules> provideVersions(String zoneId) {
148     TreeMap<String, ZoneRules> map = new TreeMap<>();
149     ZoneRules rules = getRules(zoneId, false);
150     if (rules != null) {
151         map.put(versionId, rules);
152     }
153     return map;
154 }
155
156 /**
157  * Loads the rules from a DataInputStream, often in a jar file.
158  *
159  * @param dis the DataInputStream to load, not null
160  * @throws Exception if an error occurs
161  */
162 private void load(DataInputStream dis) throws Exception {
163     if (dis.readByte() != 1) {
164         throw new StreamCorruptedException("File format not recognised");
165     }
166     // group
167     String groupId = dis.readUTF();
168     if ("TZDB".equals(groupId) == false) {
169         throw new StreamCorruptedException("File format not recognised");
170     }
171     // versions
172     int versionCount = dis.readShort();
173     for (int i = 0; i < versionCount; i++) {

```



```

174     }
175     // regions
176     int regionCount = dis.readShort();
177     String[] regionArray = new String[regionCount];
178     for (int i = 0; i < regionCount; i++) {
179         regionArray[i] = dis.readUTF();
180     }
181     regionIds = Arrays.asList(regionArray);
182     // rules
183     int ruleCount = dis.readShort();
184     Object[] ruleArray = new Object[ruleCount];
185     for (int i = 0; i < ruleCount; i++) {
186         byte[] bytes = new byte[dis.readShort()];
187         dis.readFully(bytes);
188         ruleArray[i] = bytes;
189     }
190     // link version-region-rules
191     for (int i = 0; i < versionCount; i++) {
192         int versionRegionCount = dis.readShort();
193         regionToRules.clear();
194         for (int j = 0; j < versionRegionCount; j++) {
195             String region = regionArray[dis.readShort()];
196             Object rule = ruleArray[dis.readShort() & 0xffff];
197             regionToRules.put(region, rule);
198         }
199     }
200 }
201
202 @Override
203 public String toString() {
204     return "TZDB[" + versionId + "]";
205 }
206 }

```

8.3 ZoneOffsetTransition.java

Secuencia 131: java/time/zone/ZoneOffsetTransition.java

```

1  /*
2  * Copyright (c) 2012, 2015, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *

```

```
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26 /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2009-2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62 package java.time.zone;
63
64 import java.io.DataInput;
65 import java.io.DataOutput;
66 import java.io.IOException;
67 import java.io.InvalidObjectException;
68 import java.io.ObjectInputStream;
69 import java.io.Serializable;
```

```

70 import java.time.Duration;
71 import java.time.Instant;
72 import java.time.LocalDateTime;
73 import java.time.ZoneOffset;
74 import java.util.Arrays;
75 import java.util.Collections;
76 import java.util.List;
77 import java.util.Objects;
78
79 /**
80  * A transition between two offsets caused by a discontinuity in the local time-line.
81  * <p>
82  * A transition between two offsets is normally the result of a daylight savings cutover.
83  * The discontinuity is normally a gap in spring and an overlap in autumn.
84  * {@code ZoneOffsetTransition} models the transition between the two offsets.
85  * <p>
86  * Gaps occur where there are local date-times that simply do not exist.
87  * An example would be when the offset changes from {@code +03:00} to {@code +04:00}.
88  * This might be described as 'the clocks will move forward one hour tonight at 1am'.
89  * <p>
90  * Overlaps occur where there are local date-times that exist twice.
91  * An example would be when the offset changes from {@code +04:00} to {@code +03:00}.
92  * This might be described as 'the clocks will move back one hour tonight at 2am'.
93  *
94  * @implSpec
95  * This class is immutable and thread-safe.
96  *
97  * @since 1.8
98  */
99 public final class ZoneOffsetTransition
100     implements Comparable<ZoneOffsetTransition>, Serializable {
101
102     /**
103      * Serialization version.
104      */
105     private static final long serialVersionUID = -6946044323557704546L;
106
107     /**
108      * The local transition date-time at the transition.
109      */
110     private final LocalDateTime transition;
111
112     /**
113      * The offset before transition.
114      */
115     private final ZoneOffset offsetBefore;
116
117     /**
118      * The offset after transition.
119      */
120     private final ZoneOffset offsetAfter;
121
122     /**
123      * Obtains an instance defining a transition between two offsets.
124      * <p>

```

```

123     * Applications should normally obtain an instance from {@link ZoneRules}.
124     * This factory is only intended for use when creating {@link ZoneRules}.
125     *
126     * @param transition the transition date-time at the transition, which never
127     *   actually occurs, expressed local to the before offset, not null
128     * @param offsetBefore the offset before the transition, not null
129     * @param offsetAfter the offset at and after the transition, not null
130     * @return the transition, not null
131     * @throws IllegalArgumentException if {@code offsetBefore} and {@code offsetAfter}
132     *   are equal, or {@code transition.getNano()} returns non-zero value
133     */
134     public static ZoneOffsetTransition of(LocalDateTime transition, ZoneOffset offsetBefore,
135         ZoneOffset offsetAfter) {
136         Objects.requireNonNull(transition, "transition");
137         Objects.requireNonNull(offsetBefore, "offsetBefore");
138         Objects.requireNonNull(offsetAfter, "offsetAfter");
139         if (offsetBefore.equals(offsetAfter)) {
140             throw new IllegalArgumentException("Offsets must not be equal");
141         }
142         if (transition.getNano() != 0) {
143             throw new IllegalArgumentException("Nano-of-second must be zero");
144         }
145         return new ZoneOffsetTransition(transition, offsetBefore, offsetAfter);
146     }
147
148     /**
149     * Creates an instance defining a transition between two offsets.
150     *
151     * @param transition the transition date-time with the offset before the transition, not null
152     * @param offsetBefore the offset before the transition, not null
153     * @param offsetAfter the offset at and after the transition, not null
154     */
155     ZoneOffsetTransition(LocalDateTime transition, ZoneOffset offsetBefore, ZoneOffset offsetAfter) {
156         {
157             this.transition = transition;
158             this.offsetBefore = offsetBefore;
159             this.offsetAfter = offsetAfter;
160         }
161     }
162
163     /**
164     * Creates an instance from epoch-second and offsets.
165     *
166     * @param epochSecond the transition epoch-second
167     * @param offsetBefore the offset before the transition, not null
168     * @param offsetAfter the offset at and after the transition, not null
169     */
170     ZoneOffsetTransition(long epochSecond, ZoneOffset offsetBefore, ZoneOffset offsetAfter) {
171         {
172             this.transition = LocalDateTime.ofEpochSecond(epochSecond, 0, offsetBefore);
173             this.offsetBefore = offsetBefore;
174             this.offsetAfter = offsetAfter;
175         }
176     }
177
178     //-----

```

```

174 /**
175  * Defend against malicious streams.
176  *
177  * @param s the stream to read
178  * @throws InvalidObjectException always
179  */
180 private void readObject(ObjectInputStream s) throws InvalidObjectException {
181     throw new InvalidObjectException("Deserialization via serialization delegate");
182 }
183
184 /**
185  * Writes the object using a
186  * <a href="../../serialized-form.html#java.time.zone.Ser">dedicated serialized form</a>.
187  * @serialData
188  * Refer to the serialized form of
189  * <a href="../../serialized-form.html#java.time.zone.ZoneRules">ZoneRules.writeReplace</a>
190  * for the encoding of epoch seconds and offsets.
191  * <pre style="font-size:1.0em">{@code
192  *
193  *     out.writeByte(2);           // identifies a ZoneOffsetTransition
194  *     out.writeEpochSec(toEpochSecond());
195  *     out.writeOffset(offsetBefore);
196  *     out.writeOffset(offsetAfter);
197  * }
198  * </pre>
199  * @return the replacing object, not null
200  */
201 private Object writeReplace() {
202     return new Ser(Ser.ZOT, this);
203 }
204
205 /**
206  * Writes the state to the stream.
207  *
208  * @param out the output stream, not null
209  * @throws IOException if an error occurs
210  */
211 void writeExternal(DataOutput out) throws IOException {
212     Ser.writeEpochSec(toEpochSecond(), out);
213     Ser.writeOffset(offsetBefore, out);
214     Ser.writeOffset(offsetAfter, out);
215 }
216
217 /**
218  * Reads the state from the stream.
219  *
220  * @param in the input stream, not null
221  * @return the created object, not null
222  * @throws IOException if an error occurs
223  */
224 static ZoneOffsetTransition readExternal(DataInput in) throws IOException {
225     long epochSecond = Ser.readEpochSec(in);
226     ZoneOffset before = Ser.readOffset(in);

```

```

227     ZoneOffset after = Ser.readOffset(in);
228     if (before.equals(after)) {
229         throw new IllegalArgumentException("Offsets must not be equal");
230     }
231     return new ZoneOffsetTransition(epochSecond, before, after);
232 }
233
234 //-----
235 /**
236  * Gets the transition instant.
237  * <p>
238  * This is the instant of the discontinuity, which is defined as the first
239  * instant that the 'after' offset applies.
240  * <p>
241  * The methods {@link #getInstant()}, {@link #getDateTimeBefore()} and {@link #getDateTimeAfter
242  *   ()}
243  * all represent the same instant.
244  *
245  * @return the transition instant, not null
246  */
247 public Instant getInstant() {
248     return transition.toInstant(offsetBefore);
249 }
250
251 /**
252  * Gets the transition instant as an epoch second.
253  *
254  * @return the transition epoch second
255  */
256 public long toEpochSecond() {
257     return transition.toEpochSecond(offsetBefore);
258 }
259
260 //-----
261 /**
262  * Gets the local transition date-time, as would be expressed with the 'before' offset.
263  * <p>
264  * This is the date-time where the discontinuity begins expressed with the 'before' offset.
265  * At this instant, the 'after' offset is actually used, therefore the combination of this
266  * date-time and the 'before' offset will never occur.
267  * <p>
268  * The combination of the 'before' date-time and offset represents the same instant
269  * as the 'after' date-time and offset.
270  *
271  * @return the transition date-time expressed with the before offset, not null
272  */
273 public LocalDateTime getDateTimeBefore() {
274     return transition;
275 }
276
277 /**
278  * Gets the local transition date-time, as would be expressed with the 'after' offset.
279  * <p>

```

```
279     * This is the first date-time after the discontinuity, when the new offset applies.
280     * <p>
281     * The combination of the 'before' date-time and offset represents the same instant
282     * as the 'after' date-time and offset.
283     *
284     * @return the transition date-time expressed with the after offset, not null
285     */
286     public LocalDateTime getDateTimeAfter() {
287         return transition.plusSeconds(getDurationSeconds());
288     }
289
290     /**
291     * Gets the offset before the transition.
292     * <p>
293     * This is the offset in use before the instant of the transition.
294     *
295     * @return the offset before the transition, not null
296     */
297     public ZoneOffset getOffsetBefore() {
298         return offsetBefore;
299     }
300
301     /**
302     * Gets the offset after the transition.
303     * <p>
304     * This is the offset in use on and after the instant of the transition.
305     *
306     * @return the offset after the transition, not null
307     */
308     public ZoneOffset getOffsetAfter() {
309         return offsetAfter;
310     }
311
312     /**
313     * Gets the duration of the transition.
314     * <p>
315     * In most cases, the transition duration is one hour, however this is not always the case.
316     * The duration will be positive for a gap and negative for an overlap.
317     * Time-zones are second-based, so the nanosecond part of the duration will be zero.
318     *
319     * @return the duration of the transition, positive for gaps, negative for overlaps
320     */
321     public Duration getDuration() {
322         return Duration.ofSeconds(getDurationSeconds());
323     }
324
325     /**
326     * Gets the duration of the transition in seconds.
327     *
328     * @return the duration in seconds
329     */
330     private int getDurationSeconds() {
331         return getOffsetAfter().getTotalSeconds() - getOffsetBefore().getTotalSeconds();
```

```

332     }
333
334     /**
335      * Does this transition represent a gap in the local time-line.
336      * <p>
337      * Gaps occur where there are local date-times that simply do not exist.
338      * An example would be when the offset changes from {@code +01:00} to {@code +02:00}.
339      * This might be described as 'the clocks will move forward one hour tonight at 1am'.
340      *
341      * @return true if this transition is a gap, false if it is an overlap
342      */
343     public boolean isGap() {
344         return getOffsetAfter().getTotalSeconds() > getOffsetBefore().getTotalSeconds();
345     }
346
347     /**
348      * Does this transition represent an overlap in the local time-line.
349      * <p>
350      * Overlaps occur where there are local date-times that exist twice.
351      * An example would be when the offset changes from {@code +02:00} to {@code +01:00}.
352      * This might be described as 'the clocks will move back one hour tonight at 2am'.
353      *
354      * @return true if this transition is an overlap, false if it is a gap
355      */
356     public boolean isOverlap() {
357         return getOffsetAfter().getTotalSeconds() < getOffsetBefore().getTotalSeconds();
358     }
359
360     /**
361      * Checks if the specified offset is valid during this transition.
362      * <p>
363      * This checks to see if the given offset will be valid at some point in the transition.
364      * A gap will always return false.
365      * An overlap will return true if the offset is either the before or after offset.
366      *
367      * @param offset the offset to check, null returns false
368      * @return true if the offset is valid during the transition
369      */
370     public boolean isValidOffset(ZoneOffset offset) {
371         return isGap() ? false : (getOffsetBefore().equals(offset) || getOffsetAfter().equals(
372             offset));
373     }
374
375     /**
376      * Gets the valid offsets during this transition.
377      * <p>
378      * A gap will return an empty list, while an overlap will return both offsets.
379      *
380      * @return the list of valid offsets
381      */
382     List<ZoneOffset> getValidOffsets() {
383         if (isGap()) {
384             return Collections.emptyList();
385         }
386     }

```



```

384     }
385     return Arrays.asList(getOffsetBefore(), getOffsetAfter());
386 }
387
388 //-----
389 /**
390  * Compares this transition to another based on the transition instant.
391  * <p>
392  * This compares the instants of each transition.
393  * The offsets are ignored, making this order inconsistent with equals.
394  *
395  * @param transition the transition to compare to, not null
396  * @return the comparator value, negative if less, positive if greater
397  */
398 @Override
399 public int compareTo(ZoneOffsetTransition transition) {
400     return this.getInstant().compareTo(transition.getInstant());
401 }
402
403 //-----
404 /**
405  * Checks if this object equals another.
406  * <p>
407  * The entire state of the object is compared.
408  *
409  * @param other the other object to compare to, null returns false
410  * @return true if equal
411  */
412 @Override
413 public boolean equals(Object other) {
414     if (other == this) {
415         return true;
416     }
417     if (other instanceof ZoneOffsetTransition) {
418         ZoneOffsetTransition d = (ZoneOffsetTransition) other;
419         return transition.equals(d.transition) &&
420             offsetBefore.equals(d.offsetBefore) && offsetAfter.equals(d.offsetAfter);
421     }
422     return false;
423 }
424
425 /**
426  * Returns a suitable hash code.
427  *
428  * @return the hash code
429  */
430 @Override
431 public int hashCode() {
432     return transition.hashCode() ^ offsetBefore.hashCode() ^ Integer.rotateLeft(offsetAfter.
433         hashCode(), 16);
434 }
435 //-----

```

```

436  /**
437   * Returns a string describing this object.
438   *
439   * @return a string for debugging, not null
440   */
441  @Override
442  public String toString() {
443      StringBuilder buf = new StringBuilder();
444      buf.append("Transition[")
445          .append(isGap() ? "Gap" : "Overlap")
446          .append(" at ")
447          .append(transition)
448          .append(offsetBefore)
449          .append(" to ")
450          .append(offsetAfter)
451          .append(']');
452      return buf.toString();
453  }
454
455 }
```

8.4 ZoneOffsetTransitionRule.java

Secuencia 132: java/time/zone/ZoneOffsetTransitionRule.java

```

1  /*
2   * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3   * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4   *
5   *
6   *
7   *
8   *
9   *
10  *
11  *
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  */
25
26  /*
27  *
28  *
29  *
```

```
30  *
31  *
32  * Copyright (c) 2009–2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62 package java.time.zone;
63
64 import static java.time.temporal.TemporalAdjusters.nextOrSame;
65 import static java.time.temporal.TemporalAdjusters.previousOrSame;
66
67 import java.io.DataInput;
68 import java.io.DataOutput;
69 import java.io.IOException;
70 import java.io.InvalidObjectException;
71 import java.io.ObjectInputStream;
72 import java.io.Serializable;
73 import java.time.DayOfWeek;
74 import java.time.LocalDate;
75 import java.time.LocalDateTime;
76 import java.time.LocalTime;
77 import java.time.Month;
78 import java.time.ZoneOffset;
79 import java.time.chrono.IsoChronology;
80 import java.util.Objects;
81
82 /**
```

```

83  * A rule expressing how to create a transition.
84  * <p>
85  * This class allows rules for identifying future transitions to be expressed.
86  * A rule might be written in many forms:
87  * <ul>
88  * <li>the 16th March
89  * <li>the Sunday on or after the 16th March
90  * <li>the Sunday on or before the 16th March
91  * <li>the last Sunday in February
92  * </ul>
93  * These different rule types can be expressed and queried.
94  *
95  * @implSpec
96  * This class is immutable and thread-safe.
97  *
98  * @since 1.8
99  */
100 public final class ZoneOffsetTransitionRule implements Serializable {
101
102     /**
103      * Serialization version.
104      */
105     private static final long serialVersionUID = 6889046316657758795L;
106
107     /**
108      * The month of the month-day of the first day of the cutover week.
109      * The actual date will be adjusted by the dowChange field.
110      */
111     private final Month month;
112
113     /**
114      * The day-of-month of the month-day of the cutover week.
115      * If positive, it is the start of the week where the cutover can occur.
116      * If negative, it represents the end of the week where cutover can occur.
117      * The value is the number of days from the end of the month, such that
118      * {@code -1} is the last day of the month, {@code -2} is the second
119      * to last day, and so on.
120      */
121     private final byte dom;
122
123     /**
124      * The cutover day-of-week, null to retain the day-of-month.
125      */
126     private final DayOfWeek dow;
127
128     /**
129      * The cutover time in the 'before' offset.
130      */
131     private final LocalTime time;
132
133     /**
134      * Whether the cutover time is midnight at the end of day.
135      */
136     private final boolean timeEndOfDay;
137
138     /**
139      * The definition of how the local time should be interpreted.
140      */

```

```

136     private final TimeDefinition timeDefinition;
137     /**
138      * The standard offset at the cutover.
139      */
140     private final ZoneOffset standardOffset;
141     /**
142      * The offset before the cutover.
143      */
144     private final ZoneOffset offsetBefore;
145     /**
146      * The offset after the cutover.
147      */
148     private final ZoneOffset offsetAfter;
149
150     /**
151      * Obtains an instance defining the yearly rule to create transitions between two offsets.
152      * <p>
153      * Applications should normally obtain an instance from {@link ZoneRules}.
154      * This factory is only intended for use when creating {@link ZoneRules}.
155      *
156      * @param month the month of the month-day of the first day of the cutover week, not null
157      * @param dayOfMonthIndicator the day of the month-day of the cutover week, positive if the
158      *     week is that
159      *     day or later, negative if the week is that day or earlier, counting from the last day of
160      *     the month,
161      *     from -28 to 31 excluding 0
162      * @param dayOfWeek the required day-of-week, null if the month-day should not be changed
163      * @param time the cutover time in the 'before' offset, not null
164      * @param timeEndOfDay whether the time is midnight at the end of day
165      * @param timeDefinition how to interpret the cutover
166      * @param standardOffset the standard offset in force at the cutover, not null
167      * @param offsetBefore the offset before the cutover, not null
168      * @param offsetAfter the offset after the cutover, not null
169      * @return the rule, not null
170      * @throws IllegalArgumentException if the day of month indicator is invalid
171      * @throws IllegalArgumentException if the end of day flag is true when the time is not
172      *     midnight
173      */
174     public static ZoneOffsetTransitionRule of(
175         Month month,
176         int dayOfMonthIndicator,
177         DayOfWeek dayOfWeek,
178         LocalTime time,
179         boolean timeEndOfDay,
180         TimeDefinition timeDefinition,
181         ZoneOffset standardOffset,
182         ZoneOffset offsetBefore,
183         ZoneOffset offsetAfter) {
184         Objects.requireNonNull(month, "month");
185         Objects.requireNonNull(time, "time");
186         Objects.requireNonNull(timeDefinition, "timeDefinition");
187         Objects.requireNonNull(standardOffset, "standardOffset");
188         Objects.requireNonNull(offsetBefore, "offsetBefore");
189         Objects.requireNonNull(offsetAfter, "offsetAfter");

```

```

186     Objects.requireNonNull(offsetAfter, "offsetAfter");
187     if (dayOfMonthIndicator < -28 || dayOfMonthIndicator > 31 || dayOfMonthIndicator == 0) {
188         throw new IllegalArgumentException("Day of month indicator must be between -28 and 31
189             inclusive excluding zero");
190     }
191     if (timeEndOfDay && time.equals(LocalTime.MIDNIGHT) == false) {
192         throw new IllegalArgumentException("Time must be midnight when end of day flag is true"
193             );
194     }
195     return new ZoneOffsetTransitionRule(month, dayOfMonthIndicator, dayOfWeek, time,
196         timeEndOfDay, timeDefinition, standardOffset, offsetBefore, offsetAfter);
197 }
198
199 /**
200  * Creates an instance defining the yearly rule to create transitions between two offsets.
201  *
202  * @param month the month of the month-day of the first day of the cutover week, not null
203  * @param dayOfMonthIndicator the day of the month-day of the cutover week, positive if the
204  *     week is that
205  *     day or later, negative if the week is that day or earlier, counting from the last day of
206  *     the month,
207  *     from -28 to 31 excluding 0
208  * @param dayOfWeek the required day-of-week, null if the month-day should not be changed
209  * @param time the cutover time in the 'before' offset, not null
210  * @param timeEndOfDay whether the time is midnight at the end of day
211  * @param timeDefinition how to interpret the cutover
212  * @param standardOffset the standard offset in force at the cutover, not null
213  * @param offsetBefore the offset before the cutover, not null
214  * @param offsetAfter the offset after the cutover, not null
215  * @throws IllegalArgumentException if the day of month indicator is invalid
216  * @throws IllegalArgumentException if the end of day flag is true when the time is not
217  *     midnight
218  */
219 ZoneOffsetTransitionRule(
220     Month month,
221     int dayOfMonthIndicator,
222     DayOfWeek dayOfWeek,
223     LocalTime time,
224     boolean timeEndOfDay,
225     TimeDefinition timeDefinition,
226     ZoneOffset standardOffset,
227     ZoneOffset offsetBefore,
228     ZoneOffset offsetAfter) {
229     this.month = month;
230     this.dom = (byte) dayOfMonthIndicator;
231     this.dow = dayOfWeek;
232     this.time = time;
233     this.timeEndOfDay = timeEndOfDay;
234     this.timeDefinition = timeDefinition;
235     this.standardOffset = standardOffset;
236     this.offsetBefore = offsetBefore;
237     this.offsetAfter = offsetAfter;
238 }

```

```

233
234 //-----
235 /**
236  * Defend against malicious streams.
237  *
238  * @param s the stream to read
239  * @throws InvalidObjectException always
240  */
241 private void readObject(ObjectInputStream s) throws InvalidObjectException {
242     throw new InvalidObjectException("Deserialization via serialization delegate");
243 }
244
245 /**
246  * Writes the object using a
247  * <a href="../../serialized-form.html#java.time.zone.Ser">dedicated serialized form</a>.
248  * @serialData
249  * Refer to the serialized form of
250  * <a href="../../serialized-form.html#java.time.zone.ZoneRules">ZoneRules.writeReplace</a>
251  * for the encoding of epoch seconds and offsets.
252  * <pre style="font-size:1.0em">{@code
253  *
254  *     out.writeByte(3);           // identifies a ZoneOffsetTransition
255  *     final int timeSecs = (timeEndOfDay ? 86400 : time.toSecondOfDay());
256  *     final int stdOffset = standardOffset.getTotalSeconds();
257  *     final int beforeDiff = offsetBefore.getTotalSeconds() - stdOffset;
258  *     final int afterDiff = offsetAfter.getTotalSeconds() - stdOffset;
259  *     final int timeByte = (timeSecs % 3600 == 0 ? (timeEndOfDay ? 24 : time.getHour()) : 31)
260  *
261  *     ;
262  *     final int stdOffsetByte = (stdOffset % 900 == 0 ? stdOffset / 900 + 128 : 255);
263  *     final int beforeByte = (beforeDiff == 0 || beforeDiff == 1800 || beforeDiff == 3600 ?
264  *     beforeDiff / 1800 : 3);
265  *     final int afterByte = (afterDiff == 0 || afterDiff == 1800 || afterDiff == 3600 ?
266  *     afterDiff / 1800 : 3);
267  *     final int dowByte = (dow == null ? 0 : dow.getValue());
268  *     int b = (month.getValue() << 28) +           // 4 bits
269  *             ((dom + 32) << 22) +                 // 6 bits
270  *             (dowByte << 19) +                     // 3 bits
271  *             (timeByte << 14) +                     // 5 bits
272  *             (timeDefinition.ordinal() << 12) +    // 2 bits
273  *             (stdOffsetByte << 4) +                 // 8 bits
274  *             (beforeByte << 2) +                     // 2 bits
275  *             afterByte;                             // 2 bits
276  *     out.writeInt(b);
277  *     if (timeByte == 31) {
278  *         out.writeInt(timeSecs);
279  *     }
280  *     if (stdOffsetByte == 255) {
281  *         out.writeInt(stdOffset);
282  *     }
283  *     if (beforeByte == 3) {
284  *         out.writeInt(offsetBefore.getTotalSeconds());
285  *     }
286  *     if (afterByte == 3) {

```

```

283     *         out.writeInt(offsetAfter.getTotalSeconds());
284     *     }
285     * }
286     * </pre>
287     *
288     * @return the replacing object, not null
289     */
290     private Object writeReplace() {
291         return new Ser(Ser.ZOTRULE, this);
292     }
293
294     /**
295     * Writes the state to the stream.
296     *
297     * @param out the output stream, not null
298     * @throws IOException if an error occurs
299     */
300     void writeExternal(DataOutput out) throws IOException {
301         final int timeSecs = (timeEndOfDay ? 86400 : time.toSecondOfDay());
302         final int stdOffset = standardOffset.getTotalSeconds();
303         final int beforeDiff = offsetBefore.getTotalSeconds() - stdOffset;
304         final int afterDiff = offsetAfter.getTotalSeconds() - stdOffset;
305         final int timeByte = (timeSecs % 3600 == 0 ? (timeEndOfDay ? 24 : time.getHour()) : 31);
306         final int stdOffsetByte = (stdOffset % 900 == 0 ? stdOffset / 900 + 128 : 255);
307         final int beforeByte = (beforeDiff == 0 || beforeDiff == 1800 || beforeDiff == 3600 ?
308             beforeDiff / 1800 : 3);
309         final int afterByte = (afterDiff == 0 || afterDiff == 1800 || afterDiff == 3600 ? afterDiff
310             / 1800 : 3);
311         final int dowByte = (dow == null ? 0 : dow.getValue());
312         int b = (month.getValue() << 28) + // 4 bits
313             ((dom + 32) << 22) + // 6 bits
314             (dowByte << 19) + // 3 bits
315             (timeByte << 14) + // 5 bits
316             (timeDefinition.ordinal() << 12) + // 2 bits
317             (stdOffsetByte << 4) + // 8 bits
318             (beforeByte << 2) + // 2 bits
319             afterByte; // 2 bits
320         out.writeInt(b);
321         if (timeByte == 31) {
322             out.writeInt(timeSecs);
323         }
324         if (stdOffsetByte == 255) {
325             out.writeInt(stdOffset);
326         }
327         if (beforeByte == 3) {
328             out.writeInt(offsetBefore.getTotalSeconds());
329         }
330         if (afterByte == 3) {
331             out.writeInt(offsetAfter.getTotalSeconds());
332         }
333     }
334
335     /**

```



```

334     * Reads the state from the stream.
335     *
336     * @param in the input stream, not null
337     * @return the created object, not null
338     * @throws IOException if an error occurs
339     */
340     static ZoneOffsetTransitionRule readExternal(DataInput in) throws IOException {
341         int data = in.readInt();
342         Month month = Month.of(data >>> 28);
343         int dom = ((data & (63 << 22)) >>> 22) - 32;
344         int dowByte = (data & (7 << 19)) >>> 19;
345         DayOfWeek dow = dowByte == 0 ? null : DayOfWeek.of(dowByte);
346         int timeByte = (data & (31 << 14)) >>> 14;
347         TimeDefinition defn = TimeDefinition.values()[data & (3 << 12) >>> 12];
348         int stdByte = (data & (255 << 4)) >>> 4;
349         int beforeByte = (data & (3 << 2)) >>> 2;
350         int afterByte = (data & 3);
351         LocalTime time = (timeByte == 31 ? LocalTime.ofSecondOfDay(in.readInt()) : LocalTime.of(
352             timeByte % 24, 0));
353         ZoneOffset std = (stdByte == 255 ? ZoneOffset.ofTotalSeconds(in.readInt()) : ZoneOffset.
354             ofTotalSeconds((stdByte - 128) * 900));
355         ZoneOffset before = (beforeByte == 3 ? ZoneOffset.ofTotalSeconds(in.readInt()) : ZoneOffset.
356             ofTotalSeconds(std.getTotalSeconds() + beforeByte * 1800));
357         ZoneOffset after = (afterByte == 3 ? ZoneOffset.ofTotalSeconds(in.readInt()) : ZoneOffset.
358             ofTotalSeconds(std.getTotalSeconds() + afterByte * 1800));
359         return ZoneOffsetTransitionRule.of(month, dom, dow, time, timeByte == 24, defn, std, before
360             , after);
361     }
362
363     //-----
364     /**
365     * Gets the month of the transition.
366     * <p>
367     * If the rule defines an exact date then the month is the month of that date.
368     * <p>
369     * If the rule defines a week where the transition might occur, then the month
370     * if the month of either the earliest or latest possible date of the cutover.
371     *
372     * @return the month of the transition, not null
373     */
374     public Month getMonth() {
375         return month;
376     }
377
378     /**
379     * Gets the indicator of the day-of-month of the transition.
380     * <p>
381     * If the rule defines an exact date then the day is the month of that date.
382     * <p>
383     * If the rule defines a week where the transition might occur, then the day
384     * defines either the start of the end of the transition week.
385     * <p>
386     * If the value is positive, then it represents a normal day-of-month, and is the

```

```
382     * earliest possible date that the transition can be.
383     * The date may refer to 29th February which should be treated as 1st March in non-leap years.
384     * <p>
385     * If the value is negative, then it represents the number of days back from the
386     * end of the month where {@code -1} is the last day of the month.
387     * In this case, the day identified is the latest possible date that the transition can be.
388     *
389     * @return the day-of-month indicator, from -28 to 31 excluding 0
390     */
391     public int getDayOfMonthIndicator() {
392         return dom;
393     }
394
395     /**
396     * Gets the day-of-week of the transition.
397     * <p>
398     * If the rule defines an exact date then this returns null.
399     * <p>
400     * If the rule defines a week where the cutover might occur, then this method
401     * returns the day-of-week that the month-day will be adjusted to.
402     * If the day is positive then the adjustment is later.
403     * If the day is negative then the adjustment is earlier.
404     *
405     * @return the day-of-week that the transition occurs, null if the rule defines an exact date
406     */
407     public DayOfWeek getDayOfWeek() {
408         return dow;
409     }
410
411     /**
412     * Gets the local time of day of the transition which must be checked with
413     * {@link #isMidnightEndOfDay()}.
414     * <p>
415     * The time is converted into an instant using the time definition.
416     *
417     * @return the local time of day of the transition, not null
418     */
419     public LocalTime getLocalTime() {
420         return time;
421     }
422
423     /**
424     * Is the transition local time midnight at the end of day.
425     * <p>
426     * The transition may be represented as occurring at 24:00.
427     *
428     * @return whether a local time of midnight is at the start or end of the day
429     */
430     public boolean isMidnightEndOfDay() {
431         return timeEndOfDay;
432     }
433
434     /**
```

```
435     * Gets the time definition, specifying how to convert the time to an instant.
436     * <p>
437     * The local time can be converted to an instant using the standard offset,
438     * the wall offset or UTC.
439     *
440     * @return the time definition, not null
441     */
442     public TimeDefinition getTimeDefinition() {
443         return timeDefinition;
444     }
445
446     /**
447     * Gets the standard offset in force at the transition.
448     *
449     * @return the standard offset, not null
450     */
451     public ZoneOffset getStandardOffset() {
452         return standardOffset;
453     }
454
455     /**
456     * Gets the offset before the transition.
457     *
458     * @return the offset before, not null
459     */
460     public ZoneOffset getOffsetBefore() {
461         return offsetBefore;
462     }
463
464     /**
465     * Gets the offset after the transition.
466     *
467     * @return the offset after, not null
468     */
469     public ZoneOffset getOffsetAfter() {
470         return offsetAfter;
471     }
472
473     //-----
474     /**
475     * Creates a transition instance for the specified year.
476     * <p>
477     * Calculations are performed using the ISO-8601 chronology.
478     *
479     * @param year the year to create a transition for, not null
480     * @return the transition instance, not null
481     */
482     public ZoneOffsetTransition createTransition(int year) {
483         LocalDate date;
484         if (dom < 0) {
485             date = LocalDate.of(year, month, month.length(IsoChronology.INSTANCE.isLeapYear(year))
486                 + 1 + dom);
487             if (dow != null) {
```

```

487         date = date.with(previousOrSame(dow));
488     }
489 } else {
490     date = LocalDate.of(year, month, dom);
491     if (dow != null) {
492         date = date.with(nextOrSame(dow));
493     }
494 }
495 if (timeEndOfDay) {
496     date = date.plusDays(1);
497 }
498 LocalDateTime localDT = LocalDateTime.of(date, time);
499 LocalDateTime transition = timeDefinition.createDateTime(localDT, standardOffset,
    offsetBefore);
500     return new ZoneOffsetTransition(transition, offsetBefore, offsetAfter);
501 }
502
503 //-----
504 /**
505  * Checks if this object equals another.
506  * <p>
507  * The entire state of the object is compared.
508  *
509  * @param otherRule the other object to compare to, null returns false
510  * @return true if equal
511  */
512 @Override
513 public boolean equals(Object otherRule) {
514     if (otherRule == this) {
515         return true;
516     }
517     if (otherRule instanceof ZoneOffsetTransitionRule) {
518         ZoneOffsetTransitionRule other = (ZoneOffsetTransitionRule) otherRule;
519         return month == other.month && dom == other.dom && dow == other.dow &&
520             timeDefinition == other.timeDefinition &&
521             time.equals(other.time) &&
522             timeEndOfDay == other.timeEndOfDay &&
523             standardOffset.equals(other.standardOffset) &&
524             offsetBefore.equals(other.offsetBefore) &&
525             offsetAfter.equals(other.offsetAfter);
526     }
527     return false;
528 }
529
530 /**
531  * Returns a suitable hash code.
532  *
533  * @return the hash code
534  */
535 @Override
536 public int hashCode() {
537     int hash = ((time.toSecondOfDay() + (timeEndOfDay ? 1 : 0)) << 15) +
538         (month.ordinal() << 11) + ((dom + 32) << 5) +

```

```

539         ((dow == null ? 7 : dow.ordinal()) << 2) + (timeDefinition.ordinal());
540     return hash ^ standardOffset.hashCode() ^
541         offsetBefore.hashCode() ^ offsetAfter.hashCode();
542 }
543
544 //-----
545 /**
546  * Returns a string describing this object.
547  *
548  * @return a string for debugging, not null
549  */
550 @Override
551 public String toString() {
552     StringBuilder buf = new StringBuilder();
553     buf.append("TransitionRule[")
554         .append(offsetBefore.compareTo(offsetAfter) > 0 ? "Gap " : "Overlap ")
555         .append(offsetBefore).append(" to ").append(offsetAfter).append(", ");
556     if (dow != null) {
557         if (dom == -1) {
558             buf.append(dow.name()).append(" on or before last day of ").append(month.name());
559         } else if (dom < 0) {
560             buf.append(dow.name()).append(" on or before last day minus ").append(-dom - 1).
561                 append(" of ").append(month.name());
562         } else {
563             buf.append(dow.name()).append(" on or after ").append(month.name()).append(' ').
564                 append(dom);
565         }
566     } else {
567         buf.append(month.name()).append(' ').append(dom);
568     }
569     buf.append(" at ").append(timeEndOfDay ? "24:00" : time.toString())
570         .append(" ").append(timeDefinition)
571         .append(", standard offset ").append(standardOffset)
572         .append(']');
573     return buf.toString();
574 }
575
576 //-----
577 /**
578  * A definition of the way a local time can be converted to the actual
579  * transition date-time.
580  * <p>
581  * Time zone rules are expressed in one of three ways:
582  * <ul>
583  * <li>Relative to UTC</li>
584  * <li>Relative to the standard offset in force</li>
585  * <li>Relative to the wall offset (what you would see on a clock on the wall)</li>
586  * </ul>
587  */
588 public static enum TimeDefinition {
589     /** The local date-time is expressed in terms of the UTC offset. */
590     UTC,
591     /** The local date-time is expressed in terms of the wall offset. */

```

```

590     WALL,
591     /** The local date-time is expressed in terms of the standard offset. */
592     STANDARD;
593
594     /**
595      * Converts the specified local date-time to the local date-time actually
596      * seen on a wall clock.
597      * <p>
598      * This method converts using the type of this enum.
599      * The output is defined relative to the 'before' offset of the transition.
600      * <p>
601      * The UTC type uses the UTC offset.
602      * The STANDARD type uses the standard offset.
603      * The WALL type returns the input date-time.
604      * The result is intended for use with the wall-offset.
605      *
606      * @param dateTime the local date-time, not null
607      * @param standardOffset the standard offset, not null
608      * @param wallOffset the wall offset, not null
609      * @return the date-time relative to the wall/before offset, not null
610      */
611     public LocalDateTime createDateTime(LocalDateTime dateTime, ZoneOffset standardOffset,
612         ZoneOffset wallOffset) {
613         switch (this) {
614             case UTC: {
615                 int difference = wallOffset.getTotalSeconds() - ZoneOffset.UTC.getTotalSeconds
616                     ();
617                 return dateTime.plusSeconds(difference);
618             }
619             case STANDARD: {
620                 int difference = wallOffset.getTotalSeconds() - standardOffset.getTotalSeconds
621                     ();
622                 return dateTime.plusSeconds(difference);
623             }
624             default: // WALL
625                 return dateTime;
626         }
627     }

```

8.5 ZoneRules.java

Secuencia 133: java/time/zone/ZoneRules.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *

```

```
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2009-2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
```

```
62 package java.time.zone;
63
64 import java.io.DataInput;
65 import java.io.DataOutput;
66 import java.io.IOException;
67 import java.io.InvalidObjectException;
68 import java.io.ObjectInputStream;
69 import java.io.Serializable;
70 import java.time.Duration;
71 import java.time.Instant;
72 import java.time.LocalDate;
73 import java.time.LocalDateTime;
74 import java.time.ZoneId;
75 import java.time.ZoneOffset;
76 import java.time.Year;
77 import java.util.ArrayList;
78 import java.util.Arrays;
79 import java.util.Collections;
80 import java.util.List;
81 import java.util.Objects;
82 import java.util.concurrent.ConcurrentHashMap;
83 import java.util.concurrent.ConcurrentMap;
84
85 /**
86  * The rules defining how the zone offset varies for a single time-zone.
87  * <p>
88  * The rules model all the historic and future transitions for a time-zone.
89  * {@link ZoneOffsetTransition} is used for known transitions, typically historic.
90  * {@link ZoneOffsetTransitionRule} is used for future transitions that are based
91  * on the result of an algorithm.
92  * <p>
93  * The rules are loaded via {@link ZoneRulesProvider} using a {@link ZoneId}.
94  * The same rules may be shared internally between multiple zone IDs.
95  * <p>
96  * Serializing an instance of {@code ZoneRules} will store the entire set of rules.
97  * It does not store the zone ID as it is not part of the state of this object.
98  * <p>
99  * A rule implementation may or may not store full information about historic
100  * and future transitions, and the information stored is only as accurate as
101  * that supplied to the implementation by the rules provider.
102  * Applications should treat the data provided as representing the best information
103  * available to the implementation of this rule.
104  *
105  * @implSpec
106  * This class is immutable and thread-safe.
107  *
108  * @since 1.8
109  */
110 public final class ZoneRules implements Serializable {
111
112     /**
113      * Serialization version.
114      */
```



```
115 private static final long serialVersionUID = 3044319355680032515L;
116 /**
117  * The last year to have its transitions cached.
118  */
119 private static final int LAST_CACHED_YEAR = 2100;
120
121 /**
122  * The transitions between standard offsets (epoch seconds), sorted.
123  */
124 private final long[] standardTransitions;
125 /**
126  * The standard offsets.
127  */
128 private final ZoneOffset[] standardOffsets;
129 /**
130  * The transitions between instants (epoch seconds), sorted.
131  */
132 private final long[] savingsInstantTransitions;
133 /**
134  * The transitions between local date-times, sorted.
135  * This is a paired array, where the first entry is the start of the transition
136  * and the second entry is the end of the transition.
137  */
138 private final LocalDateTime[] savingsLocalTransitions;
139 /**
140  * The wall offsets.
141  */
142 private final ZoneOffset[] wallOffsets;
143 /**
144  * The last rule.
145  */
146 private final ZoneOffsetTransitionRule[] lastRules;
147 /**
148  * The map of recent transitions.
149  */
150 private final transient ConcurrentMap<Integer, ZoneOffsetTransition[]> lastRulesCache =
151     new ConcurrentHashMap<Integer, ZoneOffsetTransition[]>();
152 /**
153  * The zero-length long array.
154  */
155 private static final long[] EMPTY_LONG_ARRAY = new long[0];
156 /**
157  * The zero-length lastrules array.
158  */
159 private static final ZoneOffsetTransitionRule[] EMPTY_LASTRULES =
160     new ZoneOffsetTransitionRule[0];
161 /**
162  * The zero-length ldt array.
163  */
164 private static final LocalDateTime[] EMPTY_LDT_ARRAY = new LocalDateTime[0];
165
166 /**
167  * Obtains an instance of a ZoneRules.
```

```

168      *
169      * @param baseStandardOffset the standard offset to use before legal rules were set, not null
170      * @param baseWallOffset the wall offset to use before legal rules were set, not null
171      * @param standardOffsetTransitionList the list of changes to the standard offset, not null
172      * @param transitionList the list of transitions, not null
173      * @param lastRules the recurring last rules, size 16 or less, not null
174      * @return the zone rules, not null
175      */
176      public static ZoneRules of(ZoneOffset baseStandardOffset,
177                                ZoneOffset baseWallOffset,
178                                List<ZoneOffsetTransition> standardOffsetTransitionList,
179                                List<ZoneOffsetTransition> transitionList,
180                                List<ZoneOffsetTransitionRule> lastRules) {
181          Objects.requireNonNull(baseStandardOffset, "baseStandardOffset");
182          Objects.requireNonNull(baseWallOffset, "baseWallOffset");
183          Objects.requireNonNull(standardOffsetTransitionList, "standardOffsetTransitionList");
184          Objects.requireNonNull(transitionList, "transitionList");
185          Objects.requireNonNull(lastRules, "lastRules");
186          return new ZoneRules(baseStandardOffset, baseWallOffset,
187                              standardOffsetTransitionList, transitionList, lastRules);
188      }
189
190      /**
191       * Obtains an instance of ZoneRules that has fixed zone rules.
192       *
193       * @param offset the offset this fixed zone rules is based on, not null
194       * @return the zone rules, not null
195       * @see #isFixedOffset()
196       */
197      public static ZoneRules of(ZoneOffset offset) {
198          Objects.requireNonNull(offset, "offset");
199          return new ZoneRules(offset);
200      }
201
202      /**
203       * Creates an instance.
204       *
205       * @param baseStandardOffset the standard offset to use before legal rules were set, not null
206       * @param baseWallOffset the wall offset to use before legal rules were set, not null
207       * @param standardOffsetTransitionList the list of changes to the standard offset, not null
208       * @param transitionList the list of transitions, not null
209       * @param lastRules the recurring last rules, size 16 or less, not null
210       */
211      ZoneRules(ZoneOffset baseStandardOffset,
212                ZoneOffset baseWallOffset,
213                List<ZoneOffsetTransition> standardOffsetTransitionList,
214                List<ZoneOffsetTransition> transitionList,
215                List<ZoneOffsetTransitionRule> lastRules) {
216          super();
217
218          // convert standard transitions
219
220          this.standardTransitions = new long[standardOffsetTransitionList.size()];

```

```

221
222     this.standardOffsets = new ZoneOffset[standardOffsetTransitionList.size() + 1];
223     this.standardOffsets[0] = baseStandardOffset;
224     for (int i = 0; i < standardOffsetTransitionList.size(); i++) {
225         this.standardTransitions[i] = standardOffsetTransitionList.get(i).toEpochSecond();
226         this.standardOffsets[i + 1] = standardOffsetTransitionList.get(i).getOffsetAfter();
227     }
228
229     // convert savings transitions to locals
230     List<LocalDateTime> localTransitionList = new ArrayList<>();
231     List<ZoneOffset> localTransitionOffsetList = new ArrayList<>();
232     localTransitionOffsetList.add(baseWallOffset);
233     for (ZoneOffsetTransition trans : transitionList) {
234         if (trans.isGap()) {
235             localTransitionList.add(trans.getDateTimeBefore());
236             localTransitionList.add(trans.getDateTimeAfter());
237         } else {
238             localTransitionList.add(trans.getDateTimeAfter());
239             localTransitionList.add(trans.getDateTimeBefore());
240         }
241         localTransitionOffsetList.add(trans.getOffsetAfter());
242     }
243     this.savingsLocalTransitions = localTransitionList.toArray(new LocalDateTime[
244         localTransitionList.size()]);
245     this.wallOffsets = localTransitionOffsetList.toArray(new ZoneOffset[
246         localTransitionOffsetList.size()]);
247
248     // convert savings transitions to instants
249     this.savingsInstantTransitions = new long[transitionList.size()];
250     for (int i = 0; i < transitionList.size(); i++) {
251         this.savingsInstantTransitions[i] = transitionList.get(i).toEpochSecond();
252     }
253
254     // last rules
255     if (lastRules.size() > 16) {
256         throw new IllegalArgumentException("Too many transition rules");
257     }
258     this.lastRules = lastRules.toArray(new ZoneOffsetTransitionRule[lastRules.size()]);
259 }
260
261 /**
262  * Constructor.
263  *
264  * @param standardTransitions the standard transitions, not null
265  * @param standardOffsets the standard offsets, not null
266  * @param savingsInstantTransitions the standard transitions, not null
267  * @param wallOffsets the wall offsets, not null
268  * @param lastRules the recurring last rules, size 15 or less, not null
269  */
270 private ZoneRules(long[] standardTransitions,
271                 ZoneOffset[] standardOffsets,
272                 long[] savingsInstantTransitions,
273                 ZoneOffset[] wallOffsets,

```

```

272         ZoneOffsetTransitionRule[] lastRules) {
273     super();
274
275     this.standardTransitions = standardTransitions;
276     this.standardOffsets = standardOffsets;
277     this.savingsInstantTransitions = savingsInstantTransitions;
278     this.wallOffsets = wallOffsets;
279     this.lastRules = lastRules;
280
281     if (savingsInstantTransitions.length == 0) {
282         this.savingsLocalTransitions = EMPTY_LDT_ARRAY;
283     } else {
284         // convert savings transitions to locals
285         List<LocalDateTime> localTransitionList = new ArrayList<>();
286         for (int i = 0; i < savingsInstantTransitions.length; i++) {
287             ZoneOffset before = wallOffsets[i];
288             ZoneOffset after = wallOffsets[i + 1];
289             ZoneOffsetTransition trans = new ZoneOffsetTransition(savingsInstantTransitions[i],
290                 before, after);
291             if (trans.isGap()) {
292                 localTransitionList.add(trans.getDateTimeBefore());
293                 localTransitionList.add(trans.getDateTimeAfter());
294             } else {
295                 localTransitionList.add(trans.getDateTimeAfter());
296                 localTransitionList.add(trans.getDateTimeBefore());
297             }
298         }
299         this.savingsLocalTransitions = localTransitionList.toArray(new LocalDateTime[
300             localTransitionList.size()]);
301     }
302
303     /**
304     * Creates an instance of ZoneRules that has fixed zone rules.
305     *
306     * @param offset the offset this fixed zone rules is based on, not null
307     * @return the zone rules, not null
308     * @see #isFixedOffset()
309     */
310     private ZoneRules(ZoneOffset offset) {
311         this.standardOffsets = new ZoneOffset[1];
312         this.standardOffsets[0] = offset;
313         this.standardTransitions = EMPTY_LONG_ARRAY;
314         this.savingsInstantTransitions = EMPTY_LONG_ARRAY;
315         this.savingsLocalTransitions = EMPTY_LDT_ARRAY;
316         this.wallOffsets = standardOffsets;
317         this.lastRules = EMPTY_LASTRULES;
318     }
319
320     /**
321     * Defend against malicious streams.
322     *
323     * @param s the stream to read

```

```

323     * @throws InvalidObjectException always
324     */
325     private void readObject(ObjectInputStream s) throws InvalidObjectException {
326         throw new InvalidObjectException("Deserialization via serialization delegate");
327     }
328
329     /**
330     * Writes the object using a
331     * <a href="../../../../serialized-form.html#java.time.zone.Ser">dedicated serialized form</a>.
332     * @serialData
333     * <pre style="font-size:1.0em">{@code
334     *
335     *     out.writeByte(1); // identifies a ZoneRules
336     *     out.writeInt(standardTransitions.length);
337     *     for (long trans : standardTransitions) {
338     *         Ser.writeEpochSec(trans, out);
339     *     }
340     *     for (ZoneOffset offset : standardOffsets) {
341     *         Ser.writeOffset(offset, out);
342     *     }
343     *     out.writeInt(savingsInstantTransitions.length);
344     *     for (long trans : savingsInstantTransitions) {
345     *         Ser.writeEpochSec(trans, out);
346     *     }
347     *     for (ZoneOffset offset : wallOffsets) {
348     *         Ser.writeOffset(offset, out);
349     *     }
350     *     out.writeByte(lastRules.length);
351     *     for (ZoneOffsetTransitionRule rule : lastRules) {
352     *         rule.writeExternal(out);
353     *     }
354     * }
355     * </pre>
356     * <p>
357     * Epoch second values used for offsets are encoded in a variable
358     * length form to make the common cases put fewer bytes in the stream.
359     * <pre style="font-size:1.0em">{@code
360     *
361     *     static void writeEpochSec(long epochSec, DataOutput out) throws IOException {
362     *         if (epochSec >= -4575744000L && epochSec < 10413792000L && epochSec % 900 == 0) { //
363     *             quarter hours between 1825 and 2300
364     *             int store = (int) ((epochSec + 4575744000L) / 900);
365     *             out.writeByte((store >>> 16) & 255);
366     *             out.writeByte((store >>> 8) & 255);
367     *             out.writeByte(store & 255);
368     *         } else {
369     *             out.writeByte(255);
370     *             out.writeLong(epochSec);
371     *         }
372     *     }
373     * </pre>
374     * <p>

```

```

375     * ZoneOffset values are encoded in a variable length form so the
376     * common cases put fewer bytes in the stream.
377     * <pre style="font-size:1.0em">{@code
378     *
379     * static void writeOffset(ZoneOffset offset, DataOutput out) throws IOException {
380     *     final int offsetSecs = offset.getTotalSeconds();
381     *     int offsetByte = offsetSecs % 900 == 0 ? offsetSecs / 900 : 127; // compress to -72 to
382     *         +72
383     *     out.writeByte(offsetByte);
384     *     if (offsetByte == 127) {
385     *         out.writeInt(offsetSecs);
386     *     }
387     * }
388     * </pre>
389     * @return the replacing object, not null
390     */
391     private Object writeReplace() {
392         return new Ser(Ser.ZRULES, this);
393     }
394
395     /**
396     * Writes the state to the stream.
397     *
398     * @param out the output stream, not null
399     * @throws IOException if an error occurs
400     */
401     void writeExternal(DataOutput out) throws IOException {
402         out.writeInt(standardTransitions.length);
403         for (long trans : standardTransitions) {
404             Ser.writeEpochSec(trans, out);
405         }
406         for (ZoneOffset offset : standardOffsets) {
407             Ser.writeOffset(offset, out);
408         }
409         out.writeInt(savingsInstantTransitions.length);
410         for (long trans : savingsInstantTransitions) {
411             Ser.writeEpochSec(trans, out);
412         }
413         for (ZoneOffset offset : wallOffsets) {
414             Ser.writeOffset(offset, out);
415         }
416         out.writeByte(lastRules.length);
417         for (ZoneOffsetTransitionRule rule : lastRules) {
418             rule.writeExternal(out);
419         }
420     }
421
422     /**
423     * Reads the state from the stream.
424     *
425     * @param in the input stream, not null
426     * @return the created object, not null

```

```

427     * @throws IOException if an error occurs
428     */
429     static ZoneRules readExternal(DataInput in) throws IOException, ClassNotFoundException {
430         int stdSize = in.readInt();
431         long[] stdTrans = (stdSize == 0) ? EMPTY_LONG_ARRAY
432                             : new long[stdSize];
433         for (int i = 0; i < stdSize; i++) {
434             stdTrans[i] = Ser.readEpochSec(in);
435         }
436         ZoneOffset[] stdOffsets = new ZoneOffset[stdSize + 1];
437         for (int i = 0; i < stdOffsets.length; i++) {
438             stdOffsets[i] = Ser.readOffset(in);
439         }
440         int savSize = in.readInt();
441         long[] savTrans = (savSize == 0) ? EMPTY_LONG_ARRAY
442                             : new long[savSize];
443         for (int i = 0; i < savSize; i++) {
444             savTrans[i] = Ser.readEpochSec(in);
445         }
446         ZoneOffset[] savOffsets = new ZoneOffset[savSize + 1];
447         for (int i = 0; i < savOffsets.length; i++) {
448             savOffsets[i] = Ser.readOffset(in);
449         }
450         int ruleSize = in.readByte();
451         ZoneOffsetTransitionRule[] rules = (ruleSize == 0) ?
452             EMPTY_LASTRULES : new ZoneOffsetTransitionRule[ruleSize];
453         for (int i = 0; i < ruleSize; i++) {
454             rules[i] = ZoneOffsetTransitionRule.readExternal(in);
455         }
456         return new ZoneRules(stdTrans, stdOffsets, savTrans, savOffsets, rules);
457     }
458
459     /**
460     * Checks of the zone rules are fixed, such that the offset never varies.
461     *
462     * @return true if the time-zone is fixed and the offset never changes
463     */
464     public boolean isFixedOffset() {
465         return savingsInstantTransitions.length == 0;
466     }
467
468     /**
469     * Gets the offset applicable at the specified instant in these rules.
470     * <p>
471     * The mapping from an instant to an offset is simple, there is only
472     * one valid offset for each instant.
473     * This method returns that offset.
474     *
475     * @param instant the instant to find the offset for, not null, but null
476     * may be ignored if the rules have a single offset for all instants
477     * @return the offset, not null
478     */
479     public ZoneOffset getOffset(Instant instant) {

```

```

480     if (savingsInstantTransitions.length == 0) {
481         return standardOffsets[0];
482     }
483     long epochSec = instant.getEpochSecond();
484     // check if using last rules
485     if (lastRules.length > 0 &&
486         epochSec > savingsInstantTransitions[savingsInstantTransitions.length - 1]) {
487         int year = findYear(epochSec, wallOffsets[wallOffsets.length - 1]);
488         ZoneOffsetTransition[] transArray = findTransitionArray(year);
489         ZoneOffsetTransition trans = null;
490         for (int i = 0; i < transArray.length; i++) {
491             trans = transArray[i];
492             if (epochSec < trans.toEpochSecond()) {
493                 return trans.getOffsetBefore();
494             }
495         }
496         return trans.getOffsetAfter();
497     }
498
499     // using historic rules
500     int index = Arrays.binarySearch(savingsInstantTransitions, epochSec);
501     if (index < 0) {
502         // switch negative insert position to start of matched range
503         index = -index - 2;
504     }
505     return wallOffsets[index + 1];
506 }
507
508 /**
509  * Gets a suitable offset for the specified local date-time in these rules.
510  * <p>
511  * The mapping from a local date-time to an offset is not straightforward.
512  * There are three cases:
513  * <ul>
514  * <li>Normal, with one valid offset. For the vast majority of the year, the normal
515  * case applies, where there is a single valid offset for the local date-time.</li>
516  * <li>Gap, with zero valid offsets. This is when clocks jump forward typically
517  * due to the spring daylight savings change from "winter" to "summer".
518  * In a gap there are local date-time values with no valid offset.</li>
519  * <li>Overlap, with two valid offsets. This is when clocks are set back typically
520  * due to the autumn daylight savings change from "summer" to "winter".
521  * In an overlap there are local date-time values with two valid offsets.</li>
522  * </ul>
523  * Thus, for any given local date-time there can be zero, one or two valid offsets.
524  * This method returns the single offset in the Normal case, and in the Gap or Overlap
525  * case it returns the offset before the transition.
526  * <p>
527  * Since, in the case of Gap and Overlap, the offset returned is a "best" value, rather
528  * than the "correct" value, it should be treated with care. Applications that care
529  * about the correct offset should use a combination of this method,
530  * {@link #getValidOffsets(LocalDateTime)} and {@link #getTransition(LocalDateTime)}.
531  *
532  * @param localDateTime the local date-time to query, not null, but null

```



```

533     * may be ignored if the rules have a single offset for all instants
534     * @return the best available offset for the local date-time, not null
535     */
536     public ZoneOffset getOffset(LocalDateTime localDateTime) {
537         Object info = getOffsetInfo(localDateTime);
538         if (info instanceof ZoneOffsetTransition) {
539             return ((ZoneOffsetTransition) info).getOffsetBefore();
540         }
541         return (ZoneOffset) info;
542     }
543
544     /**
545     * Gets the offset applicable at the specified local date-time in these rules.
546     * <p>
547     * The mapping from a local date-time to an offset is not straightforward.
548     * There are three cases:
549     * <ul>
550     * <li>Normal, with one valid offset. For the vast majority of the year, the normal
551     * case applies, where there is a single valid offset for the local date-time.</li>
552     * <li>Gap, with zero valid offsets. This is when clocks jump forward typically
553     * due to the spring daylight savings change from "winter" to "summer".
554     * In a gap there are local date-time values with no valid offset.</li>
555     * <li>Overlap, with two valid offsets. This is when clocks are set back typically
556     * due to the autumn daylight savings change from "summer" to "winter".
557     * In an overlap there are local date-time values with two valid offsets.</li>
558     * </ul>
559     * Thus, for any given local date-time there can be zero, one or two valid offsets.
560     * This method returns that list of valid offsets, which is a list of size 0, 1 or 2.
561     * In the case where there are two offsets, the earlier offset is returned at index 0
562     * and the later offset at index 1.
563     * <p>
564     * There are various ways to handle the conversion from a {@code LocalDateTime}.
565     * One technique, using this method, would be:
566     * <pre>
567     * List<ZoneOffset> validOffsets = rules.getOffset(localDT);
568     * if (validOffsets.size() == 1) {
569     *     // Normal case: only one valid offset
570     *     zoneOffset = validOffsets.get(0);
571     * } else {
572     *     // Gap or Overlap: determine what to do from transition (which will be non-null)
573     *     ZoneOffsetTransition trans = rules.getTransition(localDT);
574     * }
575     * </pre>
576     * <p>
577     * In theory, it is possible for there to be more than two valid offsets.
578     * This would happen if clocks to be put back more than once in quick succession.
579     * This has never happened in the history of time-zones and thus has no special handling.
580     * However, if it were to happen, then the list would return more than 2 entries.
581     *
582     * @param localDateTime the local date-time to query for valid offsets, not null, but null
583     * may be ignored if the rules have a single offset for all instants
584     * @return the list of valid offsets, may be immutable, not null
585     */

```

```

586 public List<ZoneOffset> getValidOffsets(LocalDateTime localDateTime) {
587     // should probably be optimized
588     Object info = getOffsetInfo(localDateTime);
589     if (info instanceof ZoneOffsetTransition) {
590         return ((ZoneOffsetTransition) info).getValidOffsets();
591     }
592     return Collections.singletonList((ZoneOffset) info);
593 }
594
595 /**
596  * Gets the offset transition applicable at the specified local date-time in these rules.
597  * <p>
598  * The mapping from a local date-time to an offset is not straightforward.
599  * There are three cases:
600  * <ul>
601  * <li>Normal, with one valid offset. For the vast majority of the year, the normal
602  * case applies, where there is a single valid offset for the local date-time.</li>
603  * <li>Gap, with zero valid offsets. This is when clocks jump forward typically
604  * due to the spring daylight savings change from "winter" to "summer".
605  * In a gap there are local date-time values with no valid offset.</li>
606  * <li>Overlap, with two valid offsets. This is when clocks are set back typically
607  * due to the autumn daylight savings change from "summer" to "winter".
608  * In an overlap there are local date-time values with two valid offsets.</li>
609  * </ul>
610  * A transition is used to model the cases of a Gap or Overlap.
611  * The Normal case will return null.
612  * <p>
613  * There are various ways to handle the conversion from a {@code LocalDateTime}.
614  * One technique, using this method, would be:
615  * <pre>
616  * ZoneOffsetTransition trans = rules.getTransition(localDT);
617  * if (trans == null) {
618  *     // Gap or Overlap: determine what to do from transition
619  * } else {
620  *     // Normal case: only one valid offset
621  *     zoneOffset = rule.getOffset(localDT);
622  * }
623  * </pre>
624  *
625  * @param localDateTime the local date-time to query for offset transition, not null, but null
626  * may be ignored if the rules have a single offset for all instants
627  * @return the offset transition, null if the local date-time is not in transition
628  */
629 public ZoneOffsetTransition getTransition(LocalDateTime localDateTime) {
630     Object info = getOffsetInfo(localDateTime);
631     return (info instanceof ZoneOffsetTransition ? (ZoneOffsetTransition) info : null);
632 }
633
634 private Object getOffsetInfo(LocalDateTime dt) {
635     if (savingsInstantTransitions.length == 0) {
636         return standardOffsets[0];
637     }
638     // check if using last rules

```

```

639     if (lastRules.length > 0 &&
640         dt.isAfter(savingsLocalTransitions[savingsLocalTransitions.length - 1])) {
641         ZoneOffsetTransition[] transArray = findTransitionArray(dt.getYear());
642         Object info = null;
643         for (ZoneOffsetTransition trans : transArray) {
644             info = findOffsetInfo(dt, trans);
645             if (info instanceof ZoneOffsetTransition || info.equals(trans.getOffsetBefore())) {
646                 return info;
647             }
648         }
649         return info;
650     }
651
652     // using historic rules
653     int index = Arrays.binarySearch(savingsLocalTransitions, dt);
654     if (index == -1) {
655         // before first transition
656         return wallOffsets[0];
657     }
658     if (index < 0) {
659         // switch negative insert position to start of matched range
660         index = -index - 2;
661     } else if (index < savingsLocalTransitions.length - 1 &&
662         savingsLocalTransitions[index].equals(savingsLocalTransitions[index + 1])) {
663         // handle overlap immediately following gap
664         index++;
665     }
666     if ((index & 1) == 0) {
667         // gap or overlap
668         LocalDateTime dtBefore = savingsLocalTransitions[index];
669         LocalDateTime dtAfter = savingsLocalTransitions[index + 1];
670         ZoneOffset offsetBefore = wallOffsets[index / 2];
671         ZoneOffset offsetAfter = wallOffsets[index / 2 + 1];
672         if (offsetAfter.getTotalSeconds() > offsetBefore.getTotalSeconds()) {
673             // gap
674             return new ZoneOffsetTransition(dtBefore, offsetBefore, offsetAfter);
675         } else {
676             // overlap
677             return new ZoneOffsetTransition(dtAfter, offsetBefore, offsetAfter);
678         }
679     } else {
680         // normal (neither gap or overlap)
681         return wallOffsets[index / 2 + 1];
682     }
683 }
684
685 /**
686  * Finds the offset info for a local date-time and transition.
687  *
688  * @param dt the date-time, not null
689  * @param trans the transition, not null
690  * @return the offset info, not null
691  */

```

```

692     private Object findOffsetInfo(LocalDateTime dt, ZoneOffsetTransition trans) {
693         LocalDateTime localTransition = trans.getDateTimeBefore();
694         if (trans.isGap()) {
695             if (dt.isBefore(localTransition)) {
696                 return trans.getOffsetBefore();
697             }
698             if (dt.isBefore(trans.getDateTimeAfter())) {
699                 return trans;
700             } else {
701                 return trans.getOffsetAfter();
702             }
703         } else {
704             if (dt.isBefore(localTransition) == false) {
705                 return trans.getOffsetAfter();
706             }
707             if (dt.isBefore(trans.getDateTimeAfter())) {
708                 return trans.getOffsetBefore();
709             } else {
710                 return trans;
711             }
712         }
713     }
714
715     /**
716      * Finds the appropriate transition array for the given year.
717      *
718      * @param year the year, not null
719      * @return the transition array, not null
720      */
721     private ZoneOffsetTransition[] findTransitionArray(int year) {
722         Integer yearObj = year; // should use Year class, but this saves a class load
723         ZoneOffsetTransition[] transArray = lastRulesCache.get(yearObj);
724         if (transArray != null) {
725             return transArray;
726         }
727         ZoneOffsetTransitionRule[] ruleArray = lastRules;
728         transArray = new ZoneOffsetTransition[ruleArray.length];
729         for (int i = 0; i < ruleArray.length; i++) {
730             transArray[i] = ruleArray[i].createTransition(year);
731         }
732         if (year < LAST_CACHED_YEAR) {
733             lastRulesCache.putIfAbsent(yearObj, transArray);
734         }
735         return transArray;
736     }
737
738     /**
739      * Gets the standard offset for the specified instant in this zone.
740      * <p>
741      * This provides access to historic information on how the standard offset
742      * has changed over time.
743      * The standard offset is the offset before any daylight saving time is applied.
744      * This is typically the offset applicable during winter.

```

```

745     *
746     * @param instant the instant to find the offset information for, not null, but null
747     * may be ignored if the rules have a single offset for all instants
748     * @return the standard offset, not null
749     */
750     public ZoneOffset getStandardOffset(Instant instant) {
751         if (savingsInstantTransitions.length == 0) {
752             return standardOffsets[0];
753         }
754         long epochSec = instant.getEpochSecond();
755         int index = Arrays.binarySearch(standardTransitions, epochSec);
756         if (index < 0) {
757             // switch negative insert position to start of matched range
758             index = -index - 2;
759         }
760         return standardOffsets[index + 1];
761     }
762
763     /**
764     * Gets the amount of daylight savings in use for the specified instant in this zone.
765     * <p>
766     * This provides access to historic information on how the amount of daylight
767     * savings has changed over time.
768     * This is the difference between the standard offset and the actual offset.
769     * Typically the amount is zero during winter and one hour during summer.
770     * Time-zones are second-based, so the nanosecond part of the duration will be zero.
771     * <p>
772     * This default implementation calculates the duration from the
773     * {@link #getOffset(java.time.Instant) actual} and
774     * {@link #getStandardOffset(java.time.Instant) standard} offsets.
775     *
776     * @param instant the instant to find the daylight savings for, not null, but null
777     * may be ignored if the rules have a single offset for all instants
778     * @return the difference between the standard and actual offset, not null
779     */
780     public Duration getDaylightSavings(Instant instant) {
781         if (savingsInstantTransitions.length == 0) {
782             return Duration.ZERO;
783         }
784         ZoneOffset standardOffset = getStandardOffset(instant);
785         ZoneOffset actualOffset = getOffset(instant);
786         return Duration.ofSeconds(actualOffset.getTotalSeconds() - standardOffset.getTotalSeconds()
787             );
788     }
789
790     /**
791     * Checks if the specified instant is in daylight savings.
792     * <p>
793     * This checks if the standard offset and the actual offset are the same
794     * for the specified instant.
795     * If they are not, it is assumed that daylight savings is in operation.
796     * <p>
797     * This default implementation compares the {@link #getOffset(java.time.Instant) actual}

```

```

797     * and {@link #getStandardOffset(java.time.Instant) standard} offsets.
798     *
799     * @param instant the instant to find the offset information for, not null, but null
800     * may be ignored if the rules have a single offset for all instants
801     * @return the standard offset, not null
802     */
803     public boolean isDaylightSavings(Instant instant) {
804         return (getStandardOffset(instant).equals(getOffset(instant)) == false);
805     }
806
807     /**
808     * Checks if the offset date-time is valid for these rules.
809     * <p>
810     * To be valid, the local date-time must not be in a gap and the offset
811     * must match one of the valid offsets.
812     * <p>
813     * This default implementation checks if {@link #getValidOffsets(java.time.LocalDateTime)}
814     * contains the specified offset.
815     *
816     * @param localDateTime the date-time to check, not null, but null
817     * may be ignored if the rules have a single offset for all instants
818     * @param offset the offset to check, null returns false
819     * @return true if the offset date-time is valid for these rules
820     */
821     public boolean isValidOffset(LocalDateTime localDateTime, ZoneOffset offset) {
822         return getValidOffsets(localDateTime).contains(offset);
823     }
824
825     /**
826     * Gets the next transition after the specified instant.
827     * <p>
828     * This returns details of the next transition after the specified instant.
829     * For example, if the instant represents a point where "Summer" daylight savings time
830     * applies, then the method will return the transition to the next "Winter" time.
831     *
832     * @param instant the instant to get the next transition after, not null, but null
833     * may be ignored if the rules have a single offset for all instants
834     * @return the next transition after the specified instant, null if this is after the last
835     * transition
836     */
837     public ZoneOffsetTransition nextTransition(Instant instant) {
838         if (savingsInstantTransitions.length == 0) {
839             return null;
840         }
841         long epochSec = instant.getEpochSecond();
842         // check if using last rules
843         if (epochSec >= savingsInstantTransitions[savingsInstantTransitions.length - 1]) {
844             if (lastRules.length == 0) {
845                 return null;
846             }
847             // search year the instant is in
848             int year = findYear(epochSec, wallOffsets[wallOffsets.length - 1]);
849             ZoneOffsetTransition[] transArray = findTransitionArray(year);

```

```

849         for (ZoneOffsetTransition trans : transArray) {
850             if (epochSec < trans.toEpochSecond()) {
851                 return trans;
852             }
853         }
854         // use first from following year
855         if (year < Year.MAX_VALUE) {
856             transArray = findTransitionArray(year + 1);
857             return transArray[0];
858         }
859         return null;
860     }
861
862     // using historic rules
863     int index = Arrays.binarySearch(savingsInstantTransitions, epochSec);
864     if (index < 0) {
865         index = -index - 1; // switched value is the next transition
866     } else {
867         index += 1; // exact match, so need to add one to get the next
868     }
869     return new ZoneOffsetTransition(savingsInstantTransitions[index], wallOffsets[index],
870                                     wallOffsets[index + 1]);
871 }
872
873 /**
874  * Gets the previous transition before the specified instant.
875  * <p>
876  * This returns details of the previous transition after the specified instant.
877  * For example, if the instant represents a point where "summer" daylight saving time
878  * applies, then the method will return the transition from the previous "winter" time.
879  *
880  * @param instant the instant to get the previous transition after, not null, but null
881  * may be ignored if the rules have a single offset for all instants
882  * @return the previous transition after the specified instant, null if this is before the
883  first transition
884  */
885 public ZoneOffsetTransition previousTransition(Instant instant) {
886     if (savingsInstantTransitions.length == 0) {
887         return null;
888     }
889     long epochSec = instant.getEpochSecond();
890     if (instant.getNano() > 0 && epochSec < Long.MAX_VALUE) {
891         epochSec += 1; // allow rest of method to only use seconds
892     }
893
894     // check if using last rules
895     long lastHistoric = savingsInstantTransitions[savingsInstantTransitions.length - 1];
896     if (lastRules.length > 0 && epochSec > lastHistoric) {
897         // search year the instant is in
898         ZoneOffset lastHistoricOffset = wallOffsets[wallOffsets.length - 1];
899         int year = findYear(epochSec, lastHistoricOffset);
900         ZoneOffsetTransition[] transArray = findTransitionArray(year);
901         for (int i = transArray.length - 1; i >= 0; i--) {

```

```

900         if (epochSec > transArray[i].toEpochSecond()) {
901             return transArray[i];
902         }
903     }
904     // use last from preceding year
905     int lastHistoricYear = findYear(lastHistoric, lastHistoricOffset);
906     if (--year > lastHistoricYear) {
907         transArray = findTransitionArray(year);
908         return transArray[transArray.length - 1];
909     }
910     // drop through
911 }
912
913 // using historic rules
914 int index = Arrays.binarySearch(savingsInstantTransitions, epochSec);
915 if (index < 0) {
916     index = -index - 1;
917 }
918 if (index <= 0) {
919     return null;
920 }
921 return new ZoneOffsetTransition(savingsInstantTransitions[index - 1], wallOffsets[index -
    1], wallOffsets[index]);
922 }
923
924 private int findYear(long epochSecond, ZoneOffset offset) {
925     // inline for performance
926     long localSecond = epochSecond + offset.getTotalSeconds();
927     long localEpochDay = Math.floorDiv(localSecond, 86400);
928     return LocalDate.ofEpochDay(localEpochDay).getYear();
929 }
930
931 /**
932  * Gets the complete list of fully defined transitions.
933  * <p>
934  * The complete set of transitions for this rules instance is defined by this method
935  * and {@link #getTransitionRules()}. This method returns those transitions that have
936  * been fully defined. These are typically historical, but may be in the future.
937  * <p>
938  * The list will be empty for fixed offset rules and for any time-zone where there has
939  * only ever been a single offset. The list will also be empty if the transition rules are
940  * unknown.
941  *
942  * @return an immutable list of fully defined transitions, not null
943  */
944 public List<ZoneOffsetTransition> getTransitions() {
945     List<ZoneOffsetTransition> list = new ArrayList<>();
946     for (int i = 0; i < savingsInstantTransitions.length; i++) {
947         list.add(new ZoneOffsetTransition(savingsInstantTransitions[i], wallOffsets[i],
948             wallOffsets[i + 1]));
949     }
950     return Collections.unmodifiableList(list);
951 }

```



```

950
951 /**
952  * Gets the list of transition rules for years beyond those defined in the transition list.
953  * <p>
954  * The complete set of transitions for this rules instance is defined by this method
955  * and {@link #getTransitions()}. This method returns instances of {@link
956     ZoneOffsetTransitionRule}
957  * that define an algorithm for when transitions will occur.
958  * <p>
959  * For any given {@code ZoneRules}, this list contains the transition rules for years
960  * beyond those years that have been fully defined. These rules typically refer to future
961  * daylight saving time rule changes.
962  * <p>
963  * If the zone defines daylight savings into the future, then the list will normally
964  * be of size two and hold information about entering and exiting daylight savings.
965  * If the zone does not have daylight savings, or information about future changes
966  * is uncertain, then the list will be empty.
967  * <p>
968  * The list will be empty for fixed offset rules and for any time-zone where there is no
969  * daylight saving time. The list will also be empty if the transition rules are unknown.
970  *
971  * @return an immutable list of transition rules, not null
972  */
973 public List<ZoneOffsetTransitionRule> getTransitionRules() {
974     return Collections.unmodifiableList(Arrays.asList(lastRules));
975 }
976
977 /**
978  * Checks if this set of rules equals another.
979  * <p>
980  * Two rule sets are equal if they will always result in the same output
981  * for any given input instant or local date-time.
982  * Rules from two different groups may return false even if they are in fact the same.
983  * <p>
984  * This definition should result in implementations comparing their entire state.
985  *
986  * @param otherRules the other rules, null returns false
987  * @return true if this rules is the same as that specified
988  */
989 @Override
990 public boolean equals(Object otherRules) {
991     if (this == otherRules) {
992         return true;
993     }
994     if (otherRules instanceof ZoneRules) {
995         ZoneRules other = (ZoneRules) otherRules;
996         return Arrays.equals(standardTransitions, other.standardTransitions) &&
997             Arrays.equals(standardOffsets, other.standardOffsets) &&
998             Arrays.equals(savingsInstantTransitions, other.savingsInstantTransitions) &&
999             Arrays.equals(wallOffsets, other.wallOffsets) &&
1000             Arrays.equals(lastRules, other.lastRules);
1001     }
1002     return false;

```

```

1002     }
1003
1004     /**
1005      * Returns a suitable hash code given the definition of {@code #equals}.
1006      *
1007      * @return the hash code
1008      */
1009     @Override
1010     public int hashCode() {
1011         return Arrays.hashCode(standardTransitions) ^
1012             Arrays.hashCode(standardOffsets) ^
1013             Arrays.hashCode(savingsInstantTransitions) ^
1014             Arrays.hashCode(wallOffsets) ^
1015             Arrays.hashCode(lastRules);
1016     }
1017
1018     /**
1019      * Returns a string describing this object.
1020      *
1021      * @return a string for debugging, not null
1022      */
1023     @Override
1024     public String toString() {
1025         return "ZoneRules[currentStandardOffset=" + standardOffsets[standardOffsets.length - 1] + "
1026             ]";
1027     }
1028 }

```

8.6 ZoneRulesException.java

Secuencia 134: java/time/zone/ZoneRulesException.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *

```

```
22  *
23  *
24  */
25
26  /*
27  * Copyright (c) 2008-2012, Stephen Colebourne & Michael Nascimento Santos
28  *
29  * All rights reserved.
30  *
31  * Redistribution and use in source and binary forms, with or without
32  * modification, are permitted provided that the following conditions are met:
33  *
34  * * Redistributions of source code must retain the above copyright notice,
35  *   this list of conditions and the following disclaimer.
36  *
37  * * Redistributions in binary form must reproduce the above copyright notice,
38  *   this list of conditions and the following disclaimer in the documentation
39  *   and/or other materials provided with the distribution.
40  *
41  * * Neither the name of JSR-310 nor the names of its contributors
42  *   may be used to endorse or promote products derived from this software
43  *   without specific prior written permission.
44  *
45  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
46  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
47  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
48  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
49  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
50  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
51  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
52  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
53  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
54  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
55  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
56  */
57  package java.time.zone;
58
59  import java.time.DateTimeException;
60
61  /**
62   * Thrown to indicate a problem with time-zone configuration.
63   * <p>
64   * This exception is used to indicate a problems with the configured
65   * time-zone rules.
66   *
67   * @implSpec
68   * This class is intended for use in a single thread.
69   *
70   * @since 1.8
71   */
72  public class ZoneRulesException extends DateTimeException {
73
74      /**
```

```

75     * Serialization version.
76     */
77     private static final long serialVersionUID = -1632418723876261839L;
78
79     /**
80     * Constructs a new date-time exception with the specified message.
81     *
82     * @param message the message to use for this exception, may be null
83     */
84     public ZoneRulesException(String message) {
85         super(message);
86     }
87
88     /**
89     * Constructs a new date-time exception with the specified message and cause.
90     *
91     * @param message the message to use for this exception, may be null
92     * @param cause the cause of the exception, may be null
93     */
94     public ZoneRulesException(String message, Throwable cause) {
95         super(message, cause);
96     }
97
98 }

```

8.7 ZoneRulesProvider.java

Secuencia 135: java/time/zone/ZoneRulesProvider.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25

```

```
26  /*
27  *
28  *
29  *
30  *
31  *
32  * Copyright (c) 2009–2012, Stephen Colebourne & Michael Nascimento Santos
33  *
34  * All rights reserved.
35  *
36  * Redistribution and use in source and binary forms, with or without
37  * modification, are permitted provided that the following conditions are met:
38  *
39  * * Redistributions of source code must retain the above copyright notice,
40  *   this list of conditions and the following disclaimer.
41  *
42  * * Redistributions in binary form must reproduce the above copyright notice,
43  *   this list of conditions and the following disclaimer in the documentation
44  *   and/or other materials provided with the distribution.
45  *
46  * * Neither the name of JSR-310 nor the names of its contributors
47  *   may be used to endorse or promote products derived from this software
48  *   without specific prior written permission.
49  *
50  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53  * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54  * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
57  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59  * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61  */
62  package java.time.zone;
63
64  import java.security.AccessController;
65  import java.security.PrivilegedAction;
66  import java.time.ZoneId;
67  import java.time.ZonedDateTime;
68  import java.util.ArrayList;
69  import java.util.HashSet;
70  import java.util.Iterator;
71  import java.util.List;
72  import java.util.NavigableMap;
73  import java.util.Objects;
74  import java.util.ServiceConfigurationError;
75  import java.util.ServiceLoader;
76  import java.util.Set;
77  import java.util.concurrent.ConcurrentHashMap;
78  import java.util.concurrent.ConcurrentMap;
```

```

79 import java.util.concurrent.CopyOnWriteArrayList;
80
81 /**
82  * Provider of time-zone rules to the system.
83  * <p>
84  * This class manages the configuration of time-zone rules.
85  * The static methods provide the public API that can be used to manage the providers.
86  * The abstract methods provide the SPI that allows rules to be provided.
87  * <p>
88  * ZoneRulesProvider may be installed in an instance of the Java Platform as
89  * extension classes, that is, jar files placed into any of the usual extension
90  * directories. Installed providers are loaded using the service-provider loading
91  * facility defined by the {@link ServiceLoader} class. A ZoneRulesProvider
92  * identifies itself with a provider configuration file named
93  * {@code java.time.zone.ZoneRulesProvider} in the resource directory
94  * {@code META-INF/services}. The file should contain a line that specifies the
95  * fully qualified concrete zonrules-provider class name.
96  * Providers may also be made available by adding them to the class path or by
97  * registering themselves via {@link #registerProvider} method.
98  * <p>
99  * The Java virtual machine has a default provider that provides zone rules
100  * for the time-zones defined by IANA Time Zone Database (TZDB). If the system
101  * property {@code java.time.zone.DefaultZoneRulesProvider} is defined then
102  * it is taken to be the fully-qualified name of a concrete ZoneRulesProvider
103  * class to be loaded as the default provider, using the system class loader.
104  * If this system property is not defined, a system-default provider will be
105  * loaded to serve as the default provider.
106  * <p>
107  * Rules are looked up primarily by zone ID, as used by {@link ZoneId}.
108  * Only zone region IDs may be used, zone offset IDs are not used here.
109  * <p>
110  * Time-zone rules are political, thus the data can change at any time.
111  * Each provider will provide the latest rules for each zone ID, but they
112  * may also provide the history of how the rules changed.
113  *
114  * @implSpec
115  * This interface is a service provider that can be called by multiple threads.
116  * Implementations must be immutable and thread-safe.
117  * <p>
118  * Providers must ensure that once a rule has been seen by the application, the
119  * rule must continue to be available.
120  * <p>
121  * Providers are encouraged to implement a meaningful {@code toString} method.
122  * <p>
123  * Many systems would like to update time-zone rules dynamically without stopping the JVM.
124  * When examined in detail, this is a complex problem.
125  * Providers may choose to handle dynamic updates, however the default provider does not.
126  *
127  * @since 1.8
128  */
129 public abstract class ZoneRulesProvider {
130
131     /**

```

```

132     * The set of loaded providers.
133     */
134     private static final CopyOnWriteArrayList<ZoneRulesProvider> PROVIDERS = new
        CopyOnWriteArrayList<>();
135     /**
136     * The lookup from zone ID to provider.
137     */
138     private static final ConcurrentMap<String, ZoneRulesProvider> ZONES = new ConcurrentHashMap
        <>(512, 0.75f, 2);

139
140     static {
141         // if the property java.time.zone.DefaultZoneRulesProvider is
142         // set then its value is the class name of the default provider
143         final List<ZoneRulesProvider> loaded = new ArrayList<>();
144         AccessController.doPrivileged(new PrivilegedAction<Object>() {
145             public Object run() {
146                 String prop = System.getProperty("java.time.zone.DefaultZoneRulesProvider");
147                 if (prop != null) {
148                     try {
149                         Class<?> c = Class.forName(prop, true, ClassLoader.getSystemClassLoader());
150                         ZoneRulesProvider provider = ZoneRulesProvider.class.cast(c.newInstance());
151                         registerProvider(provider);
152                         loaded.add(provider);
153                     } catch (Exception x) {
154                         throw new Error(x);
155                     }
156                 } else {
157                     registerProvider(new TzdbZoneRulesProvider());
158                 }
159                 return null;
160             }
161         });
162
163         ServiceLoader<ZoneRulesProvider> sl = ServiceLoader.load(ZoneRulesProvider.class,
            ClassLoader.getSystemClassLoader());
164         Iterator<ZoneRulesProvider> it = sl.iterator();
165         while (it.hasNext()) {
166             ZoneRulesProvider provider;
167             try {
168                 provider = it.next();
169             } catch (ServiceConfigurationError ex) {
170                 if (ex.getCause() instanceof SecurityException) {
171                     continue; // ignore the security exception, try the next provider
172                 }
173                 throw ex;
174             }
175             boolean found = false;
176             for (ZoneRulesProvider p : loaded) {
177                 if (p.getClass() == provider.getClass()) {
178                     found = true;
179                 }
180             }
181             if (!found) {

```

```

182         registerProvider0(provider);
183         loaded.add(provider);
184     }
185 }
186 // CopyOnWriteList could be slow if lots of providers and each added individually
187 PROVIDERS.addAll(loaded);
188 }
189
190 //-----
191 /**
192  * Gets the set of available zone IDs.
193  * <p>
194  * These IDs are the string form of a {@link ZoneId}.
195  *
196  * @return a modifiable copy of the set of zone IDs, not null
197  */
198 public static Set<String> getAvailableZoneIds() {
199     return new HashSet<>(ZONES.keySet());
200 }
201
202 /**
203  * Gets the rules for the zone ID.
204  * <p>
205  * This returns the latest available rules for the zone ID.
206  * <p>
207  * This method relies on time-zone data provider files that are configured.
208  * These are loaded using a {@code ServiceLoader}.
209  * <p>
210  * The caching flag is designed to allow provider implementations to
211  * prevent the rules being cached in {@code ZoneId}.
212  * Under normal circumstances, the caching of zone rules is highly desirable
213  * as it will provide greater performance. However, there is a use case where
214  * the caching would not be desirable, see {@link #provideRules}.
215  *
216  * @param zoneId the zone ID as defined by {@code ZoneId}, not null
217  * @param forCaching whether the rules are being queried for caching,
218  * true if the returned rules will be cached by {@code ZoneId},
219  * false if they will be returned to the user without being cached in {@code ZoneId}
220  * @return the rules, null if {@code forCaching} is true and this
221  * is a dynamic provider that wants to prevent caching in {@code ZoneId},
222  * otherwise not null
223  * @throws ZoneRulesException if rules cannot be obtained for the zone ID
224  */
225 public static ZoneRules getRules(String zoneId, boolean forCaching) {
226     Objects.requireNonNull(zoneId, "zoneId");
227     return getProvider(zoneId).provideRules(zoneId, forCaching);
228 }
229
230 /**
231  * Gets the history of rules for the zone ID.
232  * <p>
233  * Time-zones are defined by governments and change frequently.
234  * This method allows applications to find the history of changes to the

```



```

235     * rules for a single zone ID. The map is keyed by a string, which is the
236     * version string associated with the rules.
237     * <p>
238     * The exact meaning and format of the version is provider specific.
239     * The version must follow lexicographical order, thus the returned map will
240     * be order from the oldest known rules to the newest available rules.
241     * The default 'TZDB' group uses version numbering consisting of the year
242     * followed by a letter, such as '2009e' or '2012f'.
243     * <p>
244     * Implementations must provide a result for each valid zone ID, however
245     * they do not have to provide a history of rules.
246     * Thus the map will always contain one element, and will only contain more
247     * than one element if historical rule information is available.
248     *
249     * @param zoneId the zone ID as defined by {@code ZoneId}, not null
250     * @return a modifiable copy of the history of the rules for the ID, sorted
251     *         from oldest to newest, not null
252     * @throws ZoneRulesException if history cannot be obtained for the zone ID
253     */
254     public static NavigableMap<String, ZoneRules> getVersions(String zoneId) {
255         Objects.requireNonNull(zoneId, "zoneId");
256         return getProvider(zoneId).provideVersions(zoneId);
257     }
258
259     /**
260     * Gets the provider for the zone ID.
261     *
262     * @param zoneId the zone ID as defined by {@code ZoneId}, not null
263     * @return the provider, not null
264     * @throws ZoneRulesException if the zone ID is unknown
265     */
266     private static ZoneRulesProvider getProvider(String zoneId) {
267         ZoneRulesProvider provider = ZONES.get(zoneId);
268         if (provider == null) {
269             if (ZONES.isEmpty()) {
270                 throw new ZoneRulesException("No time-zone data files registered");
271             }
272             throw new ZoneRulesException("Unknown time-zone ID: " + zoneId);
273         }
274         return provider;
275     }
276
277     //-----
278     /**
279     * Registers a zone rules provider.
280     * <p>
281     * This adds a new provider to those currently available.
282     * A provider supplies rules for one or more zone IDs.
283     * A provider cannot be registered if it supplies a zone ID that has already been
284     * registered. See the notes on time-zone IDs in {@link ZoneId}, especially
285     * the section on using the concept of a "group" to make IDs unique.
286     * <p>
287     * To ensure the integrity of time-zones already created, there is no way

```

```

288     * to deregister providers.
289     *
290     * @param provider the provider to register, not null
291     * @throws ZoneRulesException if a zone ID is already registered
292     */
293     public static void registerProvider(ZoneRulesProvider provider) {
294         Objects.requireNonNull(provider, "provider");
295         registerProvider0(provider);
296         PROVIDERS.add(provider);
297     }
298
299     /**
300     * Registers the provider.
301     *
302     * @param provider the provider to register, not null
303     * @throws ZoneRulesException if unable to complete the registration
304     */
305     private static void registerProvider0(ZoneRulesProvider provider) {
306         for (String zoneId : provider.provideZoneIds()) {
307             Objects.requireNonNull(zoneId, "zoneId");
308             ZoneRulesProvider old = ZONES.putIfAbsent(zoneId, provider);
309             if (old != null) {
310                 throw new ZoneRulesException(
311                     "Unable to register zone as one already registered with that ID: " + zoneId +
312                     ", currently loading from provider: " + provider);
313             }
314         }
315     }
316
317     /**
318     * Refreshes the rules from the underlying data provider.
319     * <p>
320     * This method allows an application to request that the providers check
321     * for any updates to the provided rules.
322     * After calling this method, the offset stored in any {@link ZonedDateTime}
323     * may be invalid for the zone ID.
324     * <p>
325     * Dynamic update of rules is a complex problem and most applications
326     * should not use this method or dynamic rules.
327     * To achieve dynamic rules, a provider implementation will have to be written
328     * as per the specification of this class.
329     * In addition, instances of {@code ZoneRules} must not be cached in the
330     * application as they will become stale. However, the boolean flag on
331     * {@link #provideRules(String, boolean)} allows provider implementations
332     * to control the caching of {@code ZoneId}, potentially ensuring that
333     * all objects in the system see the new rules.
334     * Note that there is likely to be a cost in performance of a dynamic rules
335     * provider. Note also that no dynamic rules provider is in this specification.
336     *
337     * @return true if the rules were updated
338     * @throws ZoneRulesException if an error occurs during the refresh
339     */
340     public static boolean refresh() {

```

```

341     boolean changed = false;
342     for (ZoneRulesProvider provider : PROVIDERS) {
343         changed |= provider.provideRefresh();
344     }
345     return changed;
346 }
347
348 /**
349  * Constructor.
350  */
351 protected ZoneRulesProvider() {
352 }
353
354 //-----
355 /**
356  * SPI method to get the available zone IDs.
357  * <p>
358  * This obtains the IDs that this {@code ZoneRulesProvider} provides.
359  * A provider should provide data for at least one zone ID.
360  * <p>
361  * The returned zone IDs remain available and valid for the lifetime of the application.
362  * A dynamic provider may increase the set of IDs as more data becomes available.
363  *
364  * @return the set of zone IDs being provided, not null
365  * @throws ZoneRulesException if a problem occurs while providing the IDs
366  */
367 protected abstract Set<String> provideZoneIds();
368
369 /**
370  * SPI method to get the rules for the zone ID.
371  * <p>
372  * This loads the rules for the specified zone ID.
373  * The provider implementation must validate that the zone ID is valid and
374  * available, throwing a {@code ZoneRulesException} if it is not.
375  * The result of the method in the valid case depends on the caching flag.
376  * <p>
377  * If the provider implementation is not dynamic, then the result of the
378  * method must be the non-null set of rules selected by the ID.
379  * <p>
380  * If the provider implementation is dynamic, then the flag gives the option
381  * of preventing the returned rules from being cached in {@link ZoneId}.
382  * When the flag is true, the provider is permitted to return null, where
383  * null will prevent the rules from being cached in {@code ZoneId}.
384  * When the flag is false, the provider must return non-null rules.
385  *
386  * @param zoneId the zone ID as defined by {@code ZoneId}, not null
387  * @param forCaching whether the rules are being queried for caching,
388  * true if the returned rules will be cached by {@code ZoneId},
389  * false if they will be returned to the user without being cached in {@code ZoneId}
390  * @return the rules, null if {@code forCaching} is true and this
391  * is a dynamic provider that wants to prevent caching in {@code ZoneId},
392  * otherwise not null
393  * @throws ZoneRulesException if rules cannot be obtained for the zone ID

```

```

394     */
395     protected abstract ZoneRules provideRules(String zoneId, boolean forCaching);
396
397     /**
398      * SPI method to get the history of rules for the zone ID.
399      * <p>
400      * This returns a map of historical rules keyed by a version string.
401      * The exact meaning and format of the version is provider specific.
402      * The version must follow lexicographical order, thus the returned map will
403      * be order from the oldest known rules to the newest available rules.
404      * The default 'TZDB' group uses version numbering consisting of the year
405      * followed by a letter, such as '2009e' or '2012f'.
406      * <p>
407      * Implementations must provide a result for each valid zone ID, however
408      * they do not have to provide a history of rules.
409      * Thus the map will contain at least one element, and will only contain
410      * more than one element if historical rule information is available.
411      * <p>
412      * The returned versions remain available and valid for the lifetime of the application.
413      * A dynamic provider may increase the set of versions as more data becomes available.
414      *
415      * @param zoneId the zone ID as defined by {@code ZoneId}, not null
416      * @return a modifiable copy of the history of the rules for the ID, sorted
417      *         from oldest to newest, not null
418      * @throws ZoneRulesException if history cannot be obtained for the zone ID
419      */
420     protected abstract NavigableMap<String, ZoneRules> provideVersions(String zoneId);
421
422     /**
423      * SPI method to refresh the rules from the underlying data provider.
424      * <p>
425      * This method provides the opportunity for a provider to dynamically
426      * recheck the underlying data provider to find the latest rules.
427      * This could be used to load new rules without stopping the JVM.
428      * Dynamic behavior is entirely optional and most providers do not support it.
429      * <p>
430      * This implementation returns false.
431      *
432      * @return true if the rules were updated
433      * @throws ZoneRulesException if an error occurs during the refresh
434      */
435     protected boolean provideRefresh() {
436         return false;
437     }
438
439 }

```

8.8 package-info.java

Secuencia 136: java/time/zone/package-info.java

```

1  /*
2  * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
3  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.

```

```
4  *
5  *
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 */
25
26 /*
27 *
28 *
29 *
30 *
31 *
32 * Copyright (c) 2012, Stephen Colebourne & Michael Nascimento Santos
33 *
34 * All rights reserved.
35 *
36 * Redistribution and use in source and binary forms, with or without
37 * modification, are permitted provided that the following conditions are met:
38 *
39 * * Redistributions of source code must retain the above copyright notice,
40 *   this list of conditions and the following disclaimer.
41 *
42 * * Redistributions in binary form must reproduce the above copyright notice,
43 *   this list of conditions and the following disclaimer in the documentation
44 *   and/or other materials provided with the distribution.
45 *
46 * * Neither the name of JSR-310 nor the names of its contributors
47 *   may be used to endorse or promote products derived from this software
48 *   without specific prior written permission.
49 *
50 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
51 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
52 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
53 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
54 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
55 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
56 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
```

```
57 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
58 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
59 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
60 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
61 */
62
63 /**
64 * <p>
65 * Support for time-zones and their rules.
66 * </p>
67 * <p>
68 * Daylight Saving Time and Time-Zones are concepts used by Governments to alter local time.
69 * This package provides support for time-zones, their rules and the resulting
70 * gaps and overlaps in the local time-line typically caused by Daylight Saving Time.
71 * </p>
72 *
73 * <h3>Package specification</h3>
74 * <p>
75 * Unless otherwise noted, passing a null argument to a constructor or method in any class or
76 * interface
77 * in this package will cause a {@link java.lang.NullPointerException NullPointerException} to be
78 * thrown.
79 * The Javadoc "@param" definition is used to summarise the null-behavior.
80 * The "@throws {@link java.lang.NullPointerException}" is not explicitly documented in each method
81 * .
82 * </p>
83 * <p>
84 * All calculations should check for numeric overflow and throw either an {@link java.lang.
85 * ArithmeticException}
86 * or a {@link java.time.DateTimeException}.
87 * </p>
88 * @since JDK1.8
89 */
90 package java.time.zone;
```